A parallel real-time knowledge-based system and its application to diesel engine diagnostics

Stallard, P. W. A.

Award date:
1993

Awarding institution:
University of Bath

A PARALLEL REAL-TIME KNOWLEDGE-BASED SYSTEM AND ITS APPLICATION TO DIESEL ENGINE DIAGNOSTICS

Submitted by P.W.A. Stallard, B.Sc.(Hons)
for the degree of
Doctor of Philosophy
of the University of Bath
1993

P Stallard.

Bath, September 1993

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Summary

This thesis presents the design and implementation of Grape[1], a real-time parallel knowledge based-system applicable to a wide variety of time-critical problems that require the application of domain knowledge. A detailed description of the design and implementation of Grape is given. Grape uses a distributed blackboard type architecture and allows a number of inference engines to co-operate to solve a single problem or to work concurrently on different problems. The knowledge language used to represent the domain knowledge is also described.

The scheduling problem for distributing real-time tasks on a multiprocessor system is described and a solution involving a new local guarantee algorithm coupled with a global migration policy is proposed. Further, a portable scheduling scheme is developed using an enhanced co-operative scheme. Extensive experimental results of this scheduler are included.

A design of a maritime diesel engine fault diagnosis system using Grape is presented. Grape will enable a full engine condition monitoring and fault diagnosis system to be developed that will utilise a real-time engine simulation for reference data and for testing hypotheses. Such a system could increase ship safety and lead to substantial savings in maintenance and down-time costs.

[1] General purpose ReAl-time Parallel Expert system

Acknowledgements

This work would not have been possible without the help, support and encouragement of many people. My supervisor Dr R.W. Dunn provided a great deal of support and advice, for which I am very grateful. I also thank Dr A. Shamsolmaali of Lloyd's Register, my industrial supervisor, for his help and guidance.

Thanks are also due to my colleagues in the School of Electrical Engineering, especially Mr A.R. Daniels, Mr V.S. Gott, Mr B. Ross and Mr J.M. Grzejewski who all contributed to providing a pleasant and stimulating environment in which to work. I would also like to thank Dr S.J. Charlton formally of the School of Mechanical Engineering, for his help with many diesel engine issues.

I thank Professors J.F. Eastham and A.T. Johns for their permission to study in the School of Electronic and Electrical Engineering and for the provision of facilities. This project has been funded by a Science and Engineering Research Council CASE[1] award in collaboration with Lloyd's Register of Shipping under the PAYDIRT[2] project (ESPRIT II no. 5483). This funding is gratefully acknowledged.

Special thanks are due to Nicola Thorne for all her support throughout the duration of this research. Thank you for your continued encouragement and for reading (and improving) this thesis. This work and especially this thesis would not have been completed without you—thanks!

[1] Co-operative Award in Science and Engineering.
[2] Processing Architecture Yielding Deductions In Real Time.

Contents

Chapter 1
Introduction

For both safety and economic reasons, shipping operators cannot afford failures of their ship propulsion systems at sea. Such failures can endanger the crew and cause delays that may result in lost revenue or even the loss of perishable cargo. The maintenance of maritime diesel engines is therefore an essential operation that commands significant resources. To ensure that engines will not fail during operation they are usually overhauled at regular, predetermined intervals. This process is known as planned maintenance.

Planned maintenance is scheduled according to the expected life times of engine parts. At each maintenance stop, any worn parts are replaced along with any that, although healthy now, could not be guaranteed to last until the next overhaul. The scheme is inherently inefficient due to unnecessary down-times and the premature replacement of parts, and can even introduce faults into a previously healthy engine. In addition, unforeseen failures or parts that do not last as expected can still cause engine failure and unscheduled maintenance stops.

A more efficient method for maintaining maritime diesel engines is required and shipping operators are increasingly looking to condition based maintenance schemes. Condition based maintenance involves monitoring the performance of the engine as it is running and using this data to detect when engine parts are becoming worn. Such

a system could then advise operators on the severity of the problem and appropriate action can be taken.

Automatic monitoring of diesel engines is particularly applicable to maritime engines. The cost of instrumentation and the necessary monitoring hardware is not very high but it outweighs the cost of failure for most domestic and light commercial diesel engines. For larger engines, used in locomotives and trucks, condition monitoring is useful but is usually carried out periodically using external equipment rather than installing a continuous on-line performance monitoring system. With maritime diesel engines however, the equipment cost can be easily justified by the far higher cost of failure and the additional safety factor given by continuous performance monitoring. As present trends are to reduce staffing levels as far as possible, ships increasingly rely on automation to ensure their safety.

The benefits of condition monitoring over planned maintenance have been shown by a report from B.P. Tankers Limited, substantiated by other organisations, on the success of their system [1]. The report stated that condition monitoring had reduced maintenance man hours by between 5% and 40%, reduced the cost of spares by about 75% and the savings in unscheduled down-time contributed to a 20% increase in gross profit.

1.1 Project Background

One of the major problems with on-line condition monitoring is knowing what to compare the engine data with. Typically, an engine manufacturer will supply test-bed data on expected performance but in practice this may be of little use. During typical operation, a maritime diesel engine will experience large changes in ambient conditions,

sea temperature, humidity, fuel quality etc., all of which will alter the response of the engine. The quantity of data needed to compare with the engine under all possible conditions is enormous and requires special consideration. One simple solution used by some condition monitoring systems is to construct a baseline performance map and attempt to adjust these values according to current conditions. Although this has proved useful in some situations it is very approximate and will inevitably reduce the quality of diagnosis. Another approach taken by some system manufacturers, is to set up a vast central database containing performance information over a wide range of environmental conditions. Maintaining a database of this size on-board each ship is impractical so the condition monitoring systems are equipped with expensive satellite communication systems to down-load the required performance data on-line.

Another potential source of comparison is a diesel engine simulator. A simulator, if showed to be accurate, can be made to operate at the working point of the engine and can be given the same environmental conditions and fuel quality information [2]. The simulator therefore gives complete comparison data at relatively low cost.

1.2 Diesel Engine Simulation

There has been a considerable amount of work on diesel engine simulation. Simulators are traditionally used off-line, in the design and analysis of new engines. Typically, simulators run very slowly and may take minutes or even hours to simulate a single engine cycle. The reason for the slow operation is the complexity of the thermodynamic equations used to model the diesel engine, which must be solved at each time step. The speed of current simulators, severely limits their usefulness for on-line

performance comparisons.

To solve this problem, a team at the University of Bath has been working for some years to develop a real-time diesel engine simulator [2–4]. The computing power required for this is enormous and far exceeds the current computing power delivered even by the the most powerful modern serial processors. The simulation effort has therefore been to implement the system on a parallel computer. The simulator, described in Chapter 4, has been through a number of stages of evolution to the current system implemented on a transputer based computer. This version still does not achieve real-time but is able to give engine results fast enough to be of use on-line. The parallel algorithms are now very advanced and it is not believed that algorithmic improvements could be made without unreasonable effort. However, with recent advances in processor power, it is clear that within a few years a parallel machine based on modern processors and using the algorithms already developed will be capable of achieving real-time.

1.3 Health Monitoring Systems

Condition, performance and health monitoring systems are all in operation on maritime diesel engines. These systems are generally limited in their capabilities but do provide a significant benefit over traditional engine operation.

Much of the present day condition monitoring consists of periodic checks. These may use portable instrumentation and analysis hardware, vibration monitoring or other off-line techniques such as engine oil analysis. Periodic testing is more efficient for smaller land-based installations where the cost of separate monitoring equipment for each engine would be too high. As mentioned earlier however, the higher cost of failure

of maritime engines makes these periodic techniques less justified.

Typically, condition monitoring systems only identify that a fault is present rather than diagnosing what the fault actually is. The diagnosis stage is traditionally left to an experienced engineer although the data acquired in the condition monitoring stage may be useful. A more integrated approach would be desirable.

Ideally, an on-line system would monitor engine performance continuously, logging trends and detecting when actual engine performance deviates from that expected, possibly with the aid of a real-time engine simulator. At the detection stage, a more sophisticated system would analyse the deviations, applying the knowledge of trained experienced engineers, to determine what the cause of the deviations may be. A list of possible problems, ranked according to the system's confidence in them, would then be presented along with recommendations for action to be taken.

More critically, a real-time monitoring system that could analyse the deviant engine operation could, if able to respond quickly enough, shut down the engine if the conditions could lead to catastrophic failure or endanger personnel. Such a system is shown in Figure 1.1.

1.4 Research Aims

This research aims to provide a framework for implementing an on-line real-time health monitoring and fault assessment system. A number of possible solutions were considered before deciding to use an expert system based approach. The process of both fault assessment and trend analysis requires the application of uncertain knowledge to

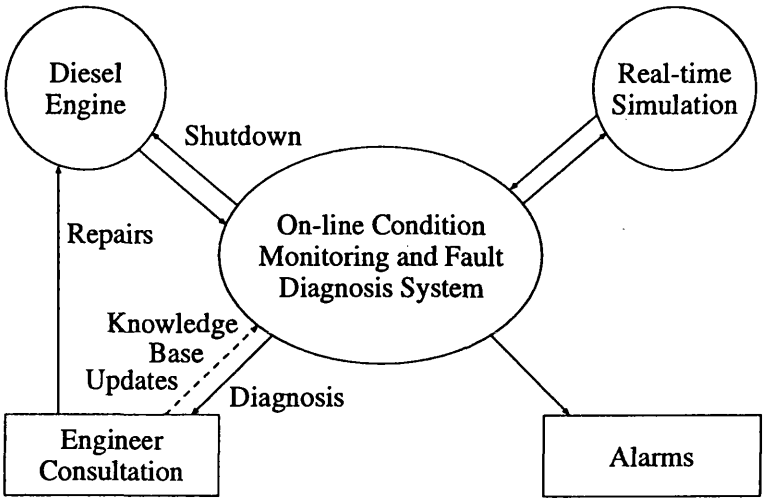

Figure 1.1: A Condition Monitoring System

uncertain data, a process unsuited to traditional algorithmic solutions. Expert systems prove to be very good at dealing with this type of problem. The use of neural networks was also considered but rejected on a number of grounds. One of the major barriers to a wider acceptance of decision support systems in general is the lack of trust operators have in these systems. Neural networks operate in a 'black box' manner producing results but without any justification. An expert system however is able to explain its reasoning and show why it believes a certain condition to be true. It is difficult to justify the use of neural networks in safety critical systems when their response to previously unseen data cannot be predicted. In addition, analysis of trends over time and the required real-time behaviour could not be provided by neural networks.

It will be shown later that current expert system technology is incapable of performing the fault assessment and health monitoring tasks without some major extensions and the emphasis of this research is to develop a system with such a capability. The system must be able to respond to external events and act upon them quickly. Using a parallel computer allows faster operation but also gives the ability to reason about

different possibilities concurrently. To exploit parallelism however requires work on parallel scheduling algorithms and other techniques such as remote locking and data sharing. Encouragement of its use in practice necessitates a flexible and portable design. As processor and operating system technology advance rapidly, any system designed without regard to portability will have a limited life time.

The aims of this research can be summarised as:

- Develop a flexible framework for on-line health monitoring and fault assessment. This requires a knowledge-based system capable of performing the diagnostic reasoning and a means of integrating this into a larger system.
- Provide real-time facilities, including the ability to react to external events and guarantee response times. This will rely on a priority based scheme to discriminate between conflicting processes.
- Develop an inference engine capable of operating in this real-time environment and dealing with the shared fact bases. The inference engine must support uncertain reasoning.
- Design a knowledge representation language (KRL) in which the knowledge can be expressed and implement the compiler to convert this language into the form required by the inference engine.
- Provide facilities for incorporating data acquisition software into the system and allow the diesel engine simulator to be used as a reference generator.
- Ensure that the design remains as independent as possible from the target hardware and operating system.

- Provide the capability of utilising multiple processors. A parallel implementation can provide both speed and functionality benefits.
- Investigate the potential uses of the simulator in the diagnostic process.

1.5 PAYDIRT

This project has been partly funded by and inspired by the ESPRIT II PAYDIRT project (Processing Architecture Yielding Deductions In Real Time) [5]. PAYDIRT involves four European partners, from Britain, France(2) and Germany, and aims to build the key tools for running real-time expert systems.

Two of the partners provide commercial application environments in which to implement and test the prototype versions of PAYDIRT. One of these is the control of the German national grid in real-time and the other is a flood water control system for Bordeaux in France.

1.6 Thesis Layout

This chapter has given a brief background of this research work and mentioned the major issues that will be addressed. The initial aims of the project are also given.

Chapter 2 will introduce the field of knowledge-based systems. Basic theory will be presented as well as a review of the current state-of-the-art. Real-time and parallel systems will be dealt with separately along with their particular problems and requirements. Previous work in these areas will also be reviewed.

Chapter 3 deals with parallel systems and real-time systems. The two subjects are covered in some detail as background to later chapters. Scheduling is described for single processor systems, parallel systems and real-time systems along with the problems currently facing the research community. A review of important work in this area is given and the chapter concludes with a look at new developments that may help to solve some of these problems in the future.

Chapter 4 describes the computing facilities used throughout this work. The main development was performed on a novel, T800 transputer based shared memory computer under an operating system called Helios. Both the system hardware and software are described. The chapter concludes with a brief history of the development of the parallel diesel engine simulator that was used during this research.

Chapter 5 describes the design of the real-time knowledge based system, known as Grape, in detail. The problem is considered and a system architecture is proposed and justified. The chapter goes on to describe the various elements of the system and highlights the major difficulties and proposed solutions. The system scheduling algorithms are derived in this chapter.

Chapter 6 details the development of the prototype system. The chapter begins with a discussion of operating system services used by the system and describes the methods used to reduce dependence on the host environment. Algorithm implementations are discussed and the implementation of the system scheduling scheme is described in detail. Techniques used for sharing data, especially with C++ and remote locking are also shown.

Chapter 7 shows how well the prototype system functioned. It includes performance measurements of the various components, such as the inference engine and the global and local schedulers. The inferencing speed of the Grape inference engine is also assessed on a range of machines. The use of Grape in parallel systems is discussed.

Chapter 8 is concerned with the implementation of a fault assessment system under the Grape system. The use of the Grape knowledge language is also demonstrated. The diagnostic system design is presented and the knowledge acquisition process is discussed. The selection of test faults is also discussed. The diesel engine simulator is also considered and the task of data acquisition is explained.

Chapter 10 assesses the current state of development and proposes work to be carried out in the future. This includes the acquisition of more detailed knowledge bases and the full integration of the system.

Chapter 11 presents the main conclusions of this research.

Chapter 2
Knowledge-Based Systems

2.1 The Background of Expert Systems

Ever since the development of the digital computer during the 1940's there have been attempts made to mimic human intelligence and reasoning. This area of computer science is known as "Artificial Intelligence" or AI and is currently seeing rapid expansion.

Many early exponents of AI made extravagant claims about the tasks computers would be able to perform in the near future. In practice, although some very impressive systems do exist, current technology is still some way short of the hopes of those early researchers. The reason for this slower rate of progress is simply that the problem in question, to mimic human intelligence, is so complex. Early research work in the area produced very little progress other than the very important realisation of the scale of the problem. It was only as a result of this unsuccessful work that researchers reassessed what they should be aiming to achieve and set more realistic, less ambitious, short-term goals.

A typical example of this early work was that started on automatic language translation in the 1950's. These systems had a number of alternative translations for each word and a set of rules that helped select, reorder and tidy up the resulting sentence. This syntactic approach led to mistakes such as the English sentence "The spirit is willing, but the flesh

is weak" being translated into Russian and back into English and emerging as "Vodka is strong, but meat is rotten" [6]. Mistakes like this occur because the approach taken is fundamentally wrong. Soon after this, it became apparent that automatic translation must involve *understanding* the language and the emphasis moved to the continuing field of natural language understanding.

Other far more fruitful work carried out in these early days of AI was concerned with general problem solving. The systems developed were capable of solving some simple problems but these had to be small and well defined. The techniques used in these domain independent problem solvers proved very valuable in other areas of AI later on. An example of one of these general problem solvers is GPS [7] developed by Ernst and Newell.

Although the generalised problem solvers did have a limited success and have gone on to under-pin much research in human cognition (e.g. SOAR [8]) they currently have very little practical use. It was soon realised that they lack the depth of knowledge that is required to solve real-world problems. The knowledge that is needed can be vast depending on the domain in question so researchers realised that if useful systems were to be built they would have to be domain specific and applied only to certain fields. The systems that began to emerge from this philosophy are known as *expert systems.*

There is no standard definition of an expert system so the term is often misused. An expert system is commonly defined as a system that performs a task that, if performed by a human would lead us to conclude that that person was an expert in that particular field. This is not really a very useful definition; a mathematician who performs complex integrations would be considered an expert but a computer program that performs the same task is not an expert system. The reason for this is in the way that the program

works. A conventional program solves a problem by applying an algorithm that is known to produce the correct answer, applying Runge-Kutta to perform integration for example, whereas an expert system works by applying domain knowledge to the problem.

Expert systems are perhaps best defined by some of their properties. In short, an expert system differs from a conventional program because it

- Reproduces human expertise.
- Has an in-depth focussed knowledge.
- Can apply uncertain knowledge and 'rules of thumb'.
- Is able to explain its behaviour and results.
- Has the ability to deal with missing or uncertain data.
- Can receive new information and knowledge without being re-programmed.

2.1.1 Early Work in Expert Systems

There are a number of very famous expert systems, DENDRAL [9], MYCIN [10], R1 [11] and PROSPECTOR [12] that were developed in the very early days of the technology. Each one has advanced the field in some respect and achieved considerable success.

DENDRAL was the first expert system to be completed, with an operational version available in 1967. It is a suite of programs that analyse the output from mass spectrometers. A mass spectrometer analyses unknown chemical compounds and produces

a distribution of molecular masses. This distribution must be interpreted by a skilled chemist who can decide what this compound was. DENDRAL started life as a tool that generated all the *possible* molecular structures that contained the elements identified by the mass spectrometer, eliminating the impossible ones by consideration of the valency etc. As the system developed, heuristic rules based on the judgement and experience of experts were added and DENDRAL now identifies the *likely* molecular structure of the compound, a job previously only undertaken by highly skilled and experienced chemists.

MYCIN, probably the most famous of all of these systems, deals with the diagnosis of bacterial infections of the blood. It pioneered the use of rules and introduced a novel way of dealing with uncertainty. It proved very successful. In a trial of 80 real life cases, MYCIN correctly diagnosed 52 of them. This compared with 50 correct answers by the best physician at Stanford, 49 by the actual therapy given and 24 by medical students. This shows just how effective MYCIN is, performing as well as an expert with years of training and far better than the less experienced doctors.

PROSPECTOR is a geological expert system. It is able to predict the presence of mineral deposits in uncharted areas by asking the user (a geologist) a number of questions about the surface appearance of the area, its soil structure etc. It was used to discover two previously unknown deposits of the mineral Molybdenum. The first was near Mount Tolman in Washington State and the second, with an estimated value of $100 million, was found in Alberta, Canada.

R1 has been used by Digital Equipment Corporation (DEC) to configure VAX computer systems for customer orders since 1982. According to DEC personnel, the use of R1 has resulted in a net saving in the order of $10–20 million per annum.

These four systems have had prolific success. The fact that they are so famous may be an indication that not all expert systems have this kind of impact but none the less, due to these early systems, expert systems are now becoming more and more common.

2.2 Expert System Theory

Expert systems can be conveniently broken down into three areas: expert knowledge stored in a **knowledge base**; an **inference engine** which uses the knowledge base to deduce new facts from the current fact base; and a **user interface** that conveys the results of the inference engine to the user. The user interface also allows the user to interrogate the inference engine about why it is asking a certain question or how it came to a certain conclusion.

2.2.1 Knowledge and its Representation

The knowledge base is the most fundamental component of any expert system. It must faithfully reflect the expertise of the domain expert and be capable of manipulation by computer. The knowledge is written in some form of *knowledge representation language* (KRL). Two types of knowledge have to be defined, factual knowledge of the type 'Bath is a city' and procedural knowledge such as 'if you are in London and wish to get to Bath, drive west along the M4'. In some cases the expert will use the KRL directly while in others a knowledge engineer will translate the expert's knowledge, obtained from interviews etc., into the KRL. The approach taken and the experience of the expert in defining his heuristic knowledge dictates the qualities required in the KRL. In most systems the KRL knowledge will be interpreted by the KBS although it may

be compiled first into a more convenient (or efficient) format. When the knowledge is used by the *knowledge based system* (KBS) it is stored in some internal representation scheme and the KRL is usually written to reflect this structure.

The knowledge representation problem is the subject of much debate and controversy. The problem was encountered long before computers were invented when the early Greek philosophers tried to find a suitable representation that allowed them to explain human psychology. With the introduction of AI however the debate has become more concerned with the practical aspects of building knowledge structures in computer memories that can then be processed by some reasoning mechanism. Many representation schemes have been proposed but most systems use one of the following: rules, semantic nets, frames and formal logic.

Most expert systems represent their knowledge in the form of rules or *productions*. The representation is the simplest to understand and implement but can still be very powerful. The knowledge is represented in a collection of '`if...`, `then...`' clauses. For example, a typical rule may be:

IF
mother(x) = y
mother(y) = z
THEN
grandmother(x) = z

This rule reads, if y is the mother of x and z is the mother of y then z is the grandmother of x. The reasoning in rule based systems is basically a pattern matching process. The expert system compares its working memory (fact base) with the antecedent clauses of each of its rules. Any rule that has all of its antecedents satisfied is said to have *fired* and the actions of the rules are performed. It is also possible to use rules the other

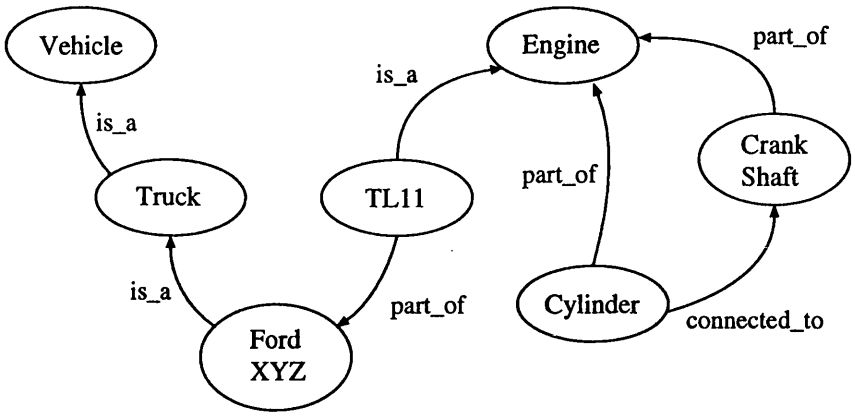

Figure 2.1: A fragment of a semantic network

way round, working from the actions to the conditions and this will be described later. Rules can represent both factual and procedural knowledge.

Semantic nets [13] are networks of objects connected by links. The objects represent elements of the problem domain and the links represent relations between those elements. Each link is a directed vector showing some form of relation. Two special links are used to represent `part_of` and `is_a` which are used to allow objects to inherit properties from other objects. A part of a semantic network is shown in Figure 2.1.

The structures are very powerful for representing what is known about a domain (the factual knowledge), but there are no formal semantics for reasoning with the networks. Typically the reasoning process will involve constructing a network about the current problem and trying to match this fragment with the knowledge base. The difficulty of reasoning with semantic networks means that they are often used to store the background knowledge of an expert system with some other representation being used for the procedural knowledge.

Frames [14] provide an object-oriented type of representation. A frame is similar to

a record structure, like those found in most data base systems, except that fields may contain procedures as well as data. Frames are therefore active data structures that can respond to and act on changes made to them. In addition to procedure members, frames may also *inherit* data from other frames and define default values. Typically, a frame will have an 'if_needed' procedure for unknown data items that define how the data can be obtained and an 'if_added' procedure that defines some action to be performed if the data item is added or changed. The combination of these procedures allows a frame based system to reason.

Formal logic is the last common knowledge representation. In most cases this will take the form of a first-order predicate logic such as that supported by Prolog. Predicate logic allows knowledge to be specified in a declarative way with clauses such as

has(engine,cylinder)

and inference rules such as

present(fuel) $\wedge$ present(air) $\wedge$ high(pressure) $\rightarrow$ ignite(fuel)

Typically reasoning will follow from a query provided by the user which will cause the logic engine to attempt to satisfy that goal using the facts and rules available. A detailed discussion of logic programming can be found in Lloyd [15] but is beyond the scope of this thesis.

All of the early systems primarily used rules for their knowledge representation although MYCIN and PROSPECTOR both used other representations as well. MYCIN used a context tree to guide the questions asked depending on the line of the reasoning so far, and PROSPECTOR used a semantic net to store its considerable background knowledge in an easily accessible and structured manner. The remainder of this chapter is concerned mainly with rule based systems although much of the content is equally

applicable to other forms of representation.

2.2.2 The Inference Engine

The inference engine is the part of the expert system that is responsible for using the knowledge to derive (or infer) new facts. The inference procedure consists of three phases, the *match* in which the rule conditions are tested against the fact base, *conflict resolution* in which a single rule is selected to fire, and the *action* when the actions of the chosen rule are performed.

Match Phase

During the match phase, the rule conditions are compared with the fact base. Each rule condition will contain either an explicit comparison such as 'is today Wednesday' which requires a simple test, or a variable binding such as 'today is the 1st of the month' which can be bound to 1st January, 1st February, 1st March etc., resulting in a set a possible bindings or *instantiations*. The rule is then able to fire if all conditions are true and there is a consistent binding for all variables. If there is more than one valid instantiation of a rule, each one will be a candidate in the conflict resolution stage. As an example, consider a rule

IF	today is in 1992
	today is the 13th of the month
	today is Friday
THEN	cancel appointments today

The instantiation set for 'today' has 366 members after the first condition, is narrowed to 12 by the second and further reduced to the two Friday 13th's of 1992 in the third.

These two instantiations are both valid instantiations of the rule and will result in the rule firing twice.

Conflict Resolution

Following the matching phase, the inference engine will have a set of all possible rule instantiations that are eligible to fire. Most inference engines at this stage perform conflict resolution to decide which rule will be allowed to fire first. This stage is necessary to avoid the problem of two instantiations attempting to add conflicting data to the fact base if they were both allowed to fire simultaneously. There are a number of strategies for deciding which instantiation should fire and the choice affects the system's sensitivity and stability. Common schemes are based on recency, which favours instantiations containing recently added data and thereby making the system more responsive, refraction which ensures the same rule is not fired twice with the same instantiation and rule ordering which uses a user defined rule priority. Other strategies such as specificity which selects the most specific eligible rule (in terms of the number of satisfied clauses and the use of variables within those clauses), can be useful under certain conditions. Most commercial expert system shells leave the final choice of conflict resolution up to the application programmer.

Inference Strategy

The overall strategy of the inference engine can be either *forward chaining* or *backward chaining*. Forward chaining systems work from the facts present in the fact base towards a conclusion. In a rule-based system for example, the known facts are compared against the conditions of the rules and when a rule fires the fact base will be altered according

to the action statements. Forward chaining is 'data driven' and is especially useful in systems where the data is available without having to ask a human user to provide it.

Backward chaining systems work from a hypothesis and try to either prove or disprove it from the available evidence. As an example, again with a rule-based system, consider the case where the inference engine wishes to investigate a certain hypothesis H. It will first find all the rules that conclude H and then go about finding if the conditions of these rules are satisfied and hence the rule will fire. If one of rule that concludes H will fire then H is proved, if none of them will then H is disproved. In the case where the evidence needed to satisfy a condition is not present, this evidence will become a new sub-goal of the inference engine and it will try to prove or disprove that. In consultation systems, such as MYCIN, where the data is acquired interactively from a user, backward chaining leads to a more coherent set of questions because of the continuity of a particular hypothesis. The choice of hypotheses from which to start backward chaining is an interesting problem. It may be decided by a priority scheme, some predefined order or perhaps the availability of certain data values. In some systems a limited forward chaining engine is used to help determine which goals will make good candidates for the main backward chaining engine.

2.2.3 State Saving Algorithms

The most time consuming step in the inferencing procedure for production systems is the matching phase. Gupta [16] has shown that the match phase can typically take around 90% of the total inferencing time and as such constitutes a major bottleneck. A class of inference algorithms has been derived to solve this problem, the first and most famous one being RETE [17].

RETE is a state-saving algorithm that exploits the facts that: only a small fraction of the current facts are changed on each cycle; and that many rules share common clauses. The algorithm works by building a data flow network of the condition clauses of the productions with nodes corresponding to comparisons of data. The top nodes represent the input data and there is a base node for each production. The results of previous comparisons are stored at each node so when the working memory is changed, only those areas of the network directly affected need to be updated. When productions are able to fire, this information is propagated to the base nodes at which point the conflict-resolution strategy selects a rule to fire.

The advantage of such state-saving networks depends on the fraction of working memory changed on each cycle and the amount of state that is saved. Gupta calculates that for his system, RETE remains efficient while less than 61% of working memory is changed on each cycle. In his experience, practical systems change less than 0.5% of working memory on each cycle giving RETE a very significant advantage over a non-state-saving algorithm.

2.2.4 The User Interface

The user interface is an important part of any system but it plays a special role in expert systems. A major factor in the widespread use of expert systems is the capability to interrogate the system via the user interface. Typically, the user will be required to enter data into the system and the interface gives the user the chance to ask why this data is being demanded. The inference procedure can be monitored and on arriving at some conclusion, the expert system can be asked to explain its reasoning. As most users of new technology are skeptical at first, the ability of the expert system to explain its

working is a significant help in the acceptance of the technology. From a development point of view, the tracing facilities also aid the knowledge base building and debugging processes.

2.2.5 Uncertainty

Expert systems are required to work on real world problems so the data, and in some cases the rules themselves may be uncertain or imprecise. A great deal of expert knowledge is of the "if A is *very* large then B *might* be C" or "if D is true then E is *more likely* to be true." An expert system must be able to deal with this sort of vagueness, along with inaccurate or faulty sensor readings etc.

There are basically two schemes that are commonly used. One is an *ad hoc* scheme developed by Shortliffe [10] for use in MYCIN and the other uses Bayes's Theorem and was first used in PROSPECTOR.

The MYCIN approach

It will be seen that with the Bayesian approach, some probabilistic analysis of domain data is necessary. Shortliffe recognised that this domain data was not available in the area in which MYCIN was to work so he devised a model of inexact reasoning more suited to his problem. The MYCIN scheme is based on the use of *certainty factors* (CF), which are associated with each fact and each rule. When a rule is evaluated, the certainty factor for the left-hand side is defined as the lowest certainty factor of all the conditions. The overall certainty factor of the rule (RCF) is then the product of the rule CF and the left-hand side CF.

A new fact created by a rule will have a certainty factor equal to the RCF. If however, a fact which already has a certainty factor of CF_1 (say), is updated then its new CF, CF_2, will be given by

$$CF_2 = \begin{cases} CF_1 + RCF - (CF_1 \times RCF) & \text{if } CF_1 > 0 \text{ and } RCF > 0 \\ CF_1 + RCF + (CF_1 \times RCF) & \text{if } CF_1 < 0 \text{ and } RCF < 0 \\ 1 & \text{if } CF_1 = -1 \text{ and } RCF = +1 \\ & \text{or } CF_1 = +1 \text{ and } RCF = -1 \\ (CF_1 + RCF) / (1 - \min(|CF_1|, |RCF|)) & \text{in all other cases} \end{cases}$$

Although this system is *ad hoc*, it does work and MYCIN is a testimony to its effectiveness. The more common approach has however been to use the Bayes's techniques as it is more mathematically rigourous.

The Bayes's Theorem Method

As mentioned, most expert systems do not use the MYCIN approach to uncertainty, favouring a method based on Bayes's Theorem. The great advantage of this technique, when the data is available, is that it encourages a statistical investigation of the domain and avoids the guess work and inevitable fine tuning of the MYCIN method.

Bayes's Theorem can be written simply as

$$P(H : E) = P(E : H) \cdot \frac{P(H)}{P(E)}$$

Thus, the probability of hypothesis H given evidence E is derived from the probability of the hypothesis, the probability of the evidence and the probability of the evidence given the hypothesis. It is important primarily because, for example, it is easier to find

the proportion of patients with measles who have spots than the proportion of people with spots who have measles.

For computational efficiency, most systems use *odds* internally rather than actual probabilities. The odds in favour of a hypothesis $O(H)$ can be calculated directly from the probabilities

$$O(H) = \frac{P(H)}{1 - P(H)}$$

This allows the odds to be updated using

$$O(H : E) = O(H) \times LR(H : E)$$

where $O(H)$ are the *prior* odds, $O(H : E)$ are the *posterior* odds and LR is the *likelihood ratio.* This equation equates the odds in favour of a hypothesis H given some evidence E to the odds in favour of H prior to the observation of E multiplied by the likelihood ratio for H given E. This is simply a transformation of Bayes's Theorem.

There are two likelihood ratios that can be calculated. The first is the *sufficiency measure, LS*

$$LS = LR(H : E) = \frac{P(E : H)}{P(E : H')}$$

where H' is not-H. The value of LS is used to calculate the posterior odds of H when E is observed. The second likelihood ratio is the *necessity measure, LN,* which is used to adjust the odds of a H when E is *not* observed. This is given by

$$LN = LR(H : E') = \frac{P(E' : H)}{P(E' : H')}$$

So, if the expert system is evaluating the hypothesis H and the evidence E is present then the odds will be adjusted using LS. If however the evidence E is not present then the odds of H will be adjusted using LN. If the evidence itself is uncertain then a linear

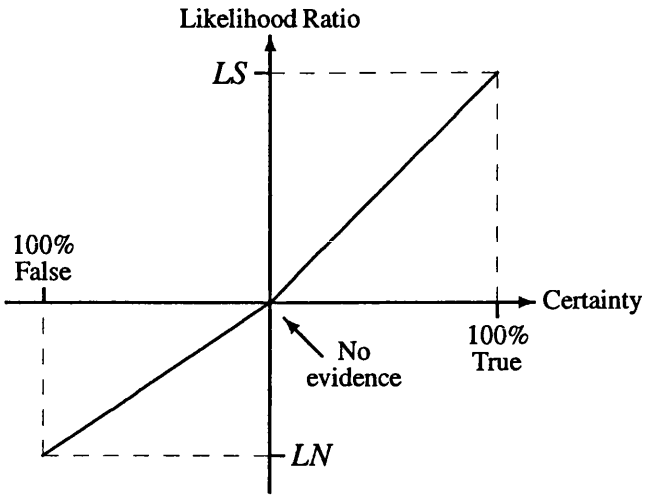

Figure 2.2: Likelihood Interpolation for Uncertain Data

interpolation should be used to calculate the value of the likelihood ratio as shown in Figure 2.2.

There are two disadvantages of the Bayesian method. The first is that the prior odds of the data must be estimated before any evidence is gathered and this depends largely on the reference sample. Should a medical system look at all patients attending the clinic, all adults in the area or the world wide population? To a large extent this can be overcome if there is sufficient evidence available to the expert system. As evidence is gathered it will point more and more towards a true or false conclusion and the value of the prior odds will be swamped. The second and more exacting problem is that the evidence must be statistically independent. This means that the knowledge engineer must be wary during the construction of the knowledge base not to use two pieces of strongly associated evidence without, for example, first combining them into a single index. Forsyth [18] gives a useful example of a weather forecasting system. It would be inadvisable for such a system to have one rule relating humidity to the probability of rain tomorrow and another rule relating cloudiness to the probability

of rain tomorrow, since cloudiness and humidity are causally connected. It would be preferable to combine the two pieces of evidence into a single "dull-dampness" index and use that in a rule predicting rainfall.

More detailed explanations of this topic can be found in Forsyth [18], Sell [19] and others.

2.3 Real-Time Knowledge Based Systems

The term "real-time" is often used to imply that a system is simply fast. Various systems claim to be real-time when in fact they concentrate entirely on increasing the efficiency and speed of the inference engines. Qi [20] for example talks about a real-time expert system when his work is concerned with the compilation of the knowledge bases in order to speed up the inference process.

There are very few real-time expert systems in use today although a number of researchers have developed experimental systems. A survey by Laffey [21] describes the current state of research and concludes that there remains a great deal to be done before real-time expert systems can be used in the countless domains in which they would be valuable.

The qualities required by a real-time expert system are more than simply speed. Unlike consultative systems, real-time systems must run continuously making the inference mechanism and memory management of the system somewhat more critical. The systems must reason non-monotonically due to the changing environments in which they operate and be able to handle asynchronous events and missing or uncertain data.

It is also essential that these systems can reason *about* time and guarantee response times. All of these points are beyond the traditional capabilities of expert systems. Speed does remain important however as a real-time expert system must be able to respond to external events on an appropriate time scale for the physical system.

Verbruggen *et al.* [22] discuss the need for on-line real-time expert systems in providing intelligent control and demonstrate a number of possible application areas. The paper also describes a practical system that uses an expert system as a controller for a process plant. The direct expert controller provides the input to the plant in the same way as a conventional digital controller but it derives the input from expert knowledge rather than a control algorithm. The concept of progressive reasoning is introduced that allows the system to continue to seek the 'correct' solution while always having a 'best so far' solution that can be presented by the deadline.

Masui *et al.* [23] deal with time-critical decision making systems, applied to air traffic control. Their system uses temporal reasoning and has the ability to schedule jobs on a priority basis.

2.4 Parallel Knowledge Based Systems

There has been an increasing number of expert systems implemented in research environments, on parallel computers. The level at which this parallelism is attained varies from parallel inference engines, through clause and rule level parallelism to systems of co-operating independent inference engines.

The parallel inference engine approach is demonstrated by Gupta [16] in his parallel

implementation of the RETE algorithm. He shows that up to 90% of the run-time of a typical expert system is spent in the match phase and therefore concentrates on the parallelisation of that. His results are reasonable but do not show dramatic speedups due to his forced serialisation during the conflict resolution and action phases. This is a typical example of Amdahl's Law [24] which states that the performance improvement of using some faster mode of execution is limited by the fraction of the time the faster mode can be used. In this case, if 10% of the execution time is performing serial operations then it is impossible to gain a speedup of more than 10 no matter how well the parallel phase performs.

Other work concentrates on detecting inter-dependencies between rules and uses this information to allocate rules to different processors. This approach, used by Ishida [25] and Li [26] allows data conflict problems to be eliminated by ensuring that rules that use the same items of data reside on the same processor.

These approaches are able to give speed improvements over serial expert systems but they do not give the system added functionality. In particular, they do not address the problems of running expert systems in real-time.

The field of distributed artificial intelligence (DAI) has been active for many years. The underlying philosophy is that many systems are naturally parallel and that a number of intelligent independent 'workers' can co-operate to solve some problems in a better way than monolithic systems. Although DAI is inherently parallel in its nature, most systems are actually serial, merely capitalising on the added functionality of co-operating systems. The communication between co-operating workers can be via direct messages or with the use of shared blackboard structures. Blackboards, first introduced by the HEARSAY [27] speech recognition system, are typically implemented in shared

memory and allow data to be shared by any number of co-operating processes.

The DAI architecture is particularly attractive in real-time environments because of the flexibility it provides. A set of independent inference engines can be scheduled according to the current system state and therefore produce a system that can respond to external events. The implementation of the system on a parallel processor also allows a number of possible solutions to be explored simultaneously allowing the best solution produced so far to be presented when required.

Other work in parallelisation has led to the development of specialised hardware architectures. Shaw's system, NON-VON [28], is a highly parallel architecture designed for the efficient implementation of large-scale knowledge-based systems that has been implemented in silicon. Another system, ZMOB [29], was designed for general AI problem solving and uses 256 nodes connected via a very high speed bus. The bus speed is fast enough to ensure that backplane contention never occurs so allows any problem geometry to be mapped to the system. DADO [30] is a massively parallel solution to AI problem solving, incorporating hundreds of thousands of relatively simple processing nodes. DADO has been used to implement an expert system using the RETE matching algorithm with impressive results. Another approach by Togai and Watanabe [31] was to fabricate an entire inference engine, capable of using fuzzy logic to cope with uncertainty, on a VLSI chip. They have used the device as an inference engine coprocessor and have achieved speeds of up to 80000 fuzzy inferences per second. Although these systems are interesting in their own right, they are inherently non-portable. The cost of such specialised hardware coupled with the uncertainty of future maintainability and support and the shortage of specialist staff make these systems unsuitable to be considered in commercial or safety critical applications.

An architecture for parallel real-time knowledge-based systems is provided by Sharma and Sridharan [32]. Their system uses a number of agents communicating through work packages that are queued to the agent that will perform them. An agenda manager is able to select the agent that each work element is directed to. They also conclude that speed-up itself is a meaningless measurement for real-time KBS and propose a set of criterion based on task throughput, average waiting times, responsiveness and processor utilisation.

2.5 Diagnostic Systems

Diagnostic systems are currently the subject of a great deal of research. When the majority of large or complex systems fail, it is still the job of a trained and experienced engineer to diagnose the faults. Current research is investigating the possibilities of automating this task but there are very few commercial systems available as yet.

There are many examples of diagnostic systems developed in research environments and the technology has been applied to a diverse range of applications from computer design diagnosis [33] to nuclear reactor core surveillance [34] and power transmission networks [35].

Diagnostic techniques vary from simple pattern recognition systems to full-blown expert systems. Analytical techniques, such as state estimation, are also used for some more predictable systems. State estimation relies on comparing measured and derived values for the system states and concluding that when the variation between them is anything but random there is a fault with the system. Another useful application of state estimation is in the prediction of sensor readings which can be useful for both

validation of sensor integrity and for the derivation of parameters which are not actually instrumented.

Some of the work on diagnostic systems has been concerned with the fundamentals of the diagnostic process itself in an attempt to formulate a generalised diagnostic framework. Keravnou and Johnson [36] use a generalised model to abstract the diagnostic process above the level of the target system. They argue that much of the diagnostic procedure is independent of what is being diagnosed so they use a modular approach that isolates the system specific areas to a single module. The model proposed by Thompson *et al.* [37] splits the diagnostic process into an initial recognition phase followed by a refinement stage that attempts to prune the search space according to a causal model that can determine feasible scenarios.

A discussion document by O'Leary [38] deals with the benefits and potential problems of diesel engine condition monitoring. His study has highlighted the savings in fuel cost, increased safety, reduced downtime and reduced maintenance costs.

Lloyd's Register of Shipping have carried out extensive research into diesel engine diagnostics using expert systems. As part of their more general work in applying information technology to all aspects of maritime fleet management [39], they have developed a system for diesel engine expert diagnosis known as DEEDS [40]. The DEEDS system is operational in demonstrable form but it does suffer from weaknesses in its real-time capabilities due to the MUSE expert system shell that was used to create it. Shamsolmaali and Banisoleiman [41] point out that the DEEDS system is incapable of reasoning over time, cannot guarantee response time, is difficult to integrate with other systems and is unable to cope with missing data. These limitations have lead them to seek a better real-time environment under which DEEDS should be implemented

and has resulted in Lloyd's supporting this research.

2.6 Summary

This chapter has introduced the field of knowledge-based systems. The basic theory of knowledge representations, uncertainty and reasoning were outlined along with some examples of early successful expert systems. The chapter went on to describe the current research areas in the field of knowledge-based systems with specific reference to real-time and parallel systems. The chapter concluded with a brief review of some of the expert diagnostic systems that have been developed around the world.

Chapter 3
Parallel and Real-Time Systems

3.1 Parallel Processing

Since its introduction in the late 1940's computing technology has improved at an incredibly fast rate. On average, the amount of work that a state-of-the-art computer can perform in a given time has more than doubled every 3 years. This has been true since the introduction of the first machines and is still true today. Modern processors are capable of performing in excess of one hundred million instructions per second and addressing giga-bytes of memory yet measure no more than 200 mm^2. Despite this, computers remain out-paced by the needs of the scientific and business communities for ever better performance.

The computing power of single microprocessors cannot continue increase at the current rate. Although it is hard to predict how soon the rate of improvement will level off, the technology is now approaching some fundamental limitations. To increase processing speed, the size of individual circuit elements can be reduced (giving faster switching times, lower latency and increased component count), the clock speed can be increased or architectural improvements such as pipelines or register windows can be introduced. In modern VLSI processors, each transistor is only a few thousand atoms wide which must, presumably, limit the reduction possible to maybe two orders of magnitude. As size decreases however, the circuit density increases and the power dissipation of the

device rises. The problem of heat dissipation is already causing problems for processor designers and this will intensify as the circuit elements become smaller. Similarly the power dissipated increases linearly with the clock speed so faster and faster clock speeds cause the same problem. There is also a limitation on clock speed due to the time required during each clock cycle for signals to travel around the device—this is fundamentally limited by the speed of light. There are already some interesting developments to help overcome some of these problems. Work in asynchronous processor design may remove the need for a central clock and also reduce some heat dissipation problems as the switching times of devices will not be tied rigidly to clock cycles. There will doubtless be other architectural changes and technological advances to help increase processor speeds in the future but there are clearly limits and these may now not be too far away. One last point is the complexity of modern processors. New generation processors are extremely complex, involving many millions of transistors and taking many person-years of development. It is unlikely that it will be possible for processors of much greater complexity to be developed at all using the CAD technology and the specialised personnel used today.

Many fields in which computers work could use any amount of computing power available. One such area is computational fluid dynamics, CFD, the modelling of complex fluid flows. CFD packages solve the flow in some volume by dividing it into a large number of cells and defining a set of boundary conditions for the volume edges. The solution then proceeds by solving the complex set of fluid dynamics equations for each cell, adjusting the flow values between cells accordingly, and repeating. Eventually, this iteration process will converge towards some solution and the process can be stopped. In practice, the accuracy of the solution is always limited by the time available to wait for the result. Present systems are still limited to relatively

small models purely because of the computational intensity of the task. As computing power increases, solutions will be available quicker, or with greater accuracy, and larger models will be solvable but there is no upper limit on the amount of power that CFD could absorb. It is not only specialist applications that are demanding faster and faster machines. The complexity of software in general is increasing and more powerful machines are required to compensate for this. Most people already expect a graphical user interface and these expectations will continue to grow with the evolution of multimedia systems etc. Machines will need to be more powerful in the future just to support the human-computer interface.

There is clearly a demand for computers to provide more power than is currently available from state-of-the-art processors. This demand will become even more acute as the rate of progress declines so some other means of providing more power will have to be found. One solution to this problem is using parallel processing which, as its name suggests, is a technique for solving problems using more than one processor at a time. It is common sense that two processors should be able to co-operate to do a job faster than a single processor but there a number of problems when this is attempted in practice. Some of these will be dealt with in this chapter.

3.1.1 Parallelisation

Parallel computing is a rapidly evolving field attracting interest from researchers, manufacturers and end users. Although parallel computers have been in existence for many years, it is only recently that the technology has begun to be exploited in commercial systems.

For more than one processor to work on a single problem requires firstly that the problem is inherently parallel. This means that there must be parts of the code that could be performed simultaneously. For example, problems in which some simple function has to be performed on each one of a set of inputs are inherently parallel. A number of these function units could run in parallel and work on different areas of the input set—given that the processing of each data item is independent.

In practice, data is not independent and parallel computers must provide a mechanism for the individual processors to communicate with each other and share data. Provided the quantity of data that must be transferred between processors is not too high compared with the level of computation it is still beneficial to parallelise problems requiring communications. As mentioned earlier, CFD solvers divide the problem into a number of interconnected cells. As the processing required by each cell is independent and the communication necessary for updating the inter-cell flow parameters at each iteration is quite low, CFD is a good example of an efficiently parallelisable problem. In practice the cost of communication would be too high to allow each cell to be calculated on a separate processor. Instead a region of cells would be partitioned on to each node thus increasing the ratio of computation to communication.

Parallel computers are usually classified as either shared memory systems or distributed memory systems according to the mechanism provided for inter-processor communication. Shared memory computers provide an area of memory that can be accessed by every processor via a shared memory bus. Most of the early parallel computers used this approach and the architecture can be very effective, especially for systems with relatively few processors. Distributed memory systems do not have any shared memory but instead communicate by passing messages through communication channels that

interconnect the processors. Both of these architectures have drawbacks so computer architects are now attempting to draw the best from both paradigms to produce the machines for the future.

3.1.2 Shared Memory Computers

Shared memory systems provide, as mentioned earlier, an area of memory that can be accessed by every processor. This may be a single global area of memory that forms the sole memory resource in the system. Every processor will have its local code and data stored in this memory along with any shared data. An architecture of this kind proves inefficient because of the very high contention for the memory. Only one processor can actually access the memory at a time so all the other processes are forced to wait for the bus to become available. This problem can be overcome in one of two ways. Either each processor can be given a local cache that will satisfy most accesses locally and therefore reduce contention on the bus or each processor can be given its own local memory to store the program code and local data. The first solution leads to potential coherency problems between the local caches in which there are multiple values associated with the same item of data. This is avoided by using a snooping mechanism on each node that can detect when another node wishes to update a data item and can remove it from its cache. Most shared memory machines available today use this type of mechanism to reduce backplane contention. The alternative approach of giving local memory to each node can work equally well but it requires more effort on the part of the programmer as the memory becomes non-uniform.

Shared memory systems, especially with snooping caches, work well in practice for machines with a limited number of processors. The processors are tightly coupled

and global memory accesses are fast. However, with high levels of inter-processor communication or with a large number of processors, the global memory bus contention quickly becomes significant and will reduce the system performance. The use of a single bus limits systems to tens of processors.

3.1.3 Distributed Memory Computers

The only interconnection between processors in a distributed memory system is some form of communication channel. This may be a high speed serial link, a local area network or even a telephone connection. As such, distributed memory computers can be physically distributed as well as logically distributed. All data and other information shared between processors is passed in messages via the communication channels. Until the recent introduction of packet-switching routing networks, it has been impossible to fully interconnect more than a small number of processors. Instead the processors had to be connected in some specified topology in which each processor was connected to only a subset of the other nodes. The interconnection topology has a crucial effect on the system performance. Many algorithms fit well on to particular topologies of processors and these often perform well on distributed memory machines. Other problems however require each processor to communicate with many other processors so messages destined for nodes that are not directly reachable must be routed via other nodes. This introduces inefficiency because a third processor has to take part in a message transfer and also greatly increases the message latency.

Distributed memory computers suffer from slow inter-processor communication compared with shared memory systems. Clearly, a memory bus is typically transferring 32 or 64 bits of information every cycle so will outperform a serial link substantially.

Another problem is that both processors must be involved, to some extent, in every message passing operation whereas only one processor has to be involved with a shared memory system. Distributed memory systems do however have some important advantages; they are more secure and most importantly more scalable.

Each time a new node is added to a shared memory bus, contention for the bus increases. The bus bandwidth is constant so the portion of that available to each node decreases linearly with the number of nodes. Each time a node is added to a distributed system however it adds extra communication bandwidth to the system so the bandwidth of the entire system grows linearly with the number of nodes.

An important programming model for distributed systems is C.A.R. Hoare's *Communicating Sequential Processes* [42]. The basic concept of CSP is that computer systems (both the hardware and software) can be divided into subsystems (processes) that operate concurrently and communicate with each other and with their environment through defined channels. CSP has a mathematical foundation that allows the behaviour of processes to be reasoned about. Using CSP it is possible to prove that a system of interacting processes will terminate and that it is deadlock free as well as that the results will be correct. A formal calculus also allows programs to be transformed (into hardware for example) while retaining the correctness proved for the original program. CSP has until recently had no notion of timeliness so its potential use in real-time systems remains limited.

3.1.4 Virtual Shared Memory (VSM)

There are several research groups and commercial manufacturers currently working on systems to support Virtual Shared Memory (VSM). These machines attempt to merge the best features of both shared memory and distributed memory machines. VSM systems provide a shared memory model of programming on top of a physically distributed memory system by means of some form of coherency protocol. The combination gives the programmer a simple shared memory view of the machine but provides the high bandwidth and scalability given by distributed networks.

3.1.5 Locality and Granularity

Locality of reference is a crucial property of any scalable parallel algorithm. A program can be said to have locality when most of the accesses made by each processor are satisfied locally. When accesses cannot be satisfied locally a communication overhead is introduced to obtain the data. The ratio of time spent communicating to the time spent processing determines the efficiency of processor usage and hence the potential benefit to be gained from using a parallel computer. This ratio is directly related to the system's 'granularity'—the level at which the problem was broken down for parallelisation. The granularity of a system would be classed as 'fine' if the task was split at a very low level, possibly as low as single instructions or as 'coarse' if the division is at a higher (functional) level. The choice of how to break down a problem on to a parallel machine is determined by several factors. It may be that the algorithm has a 'natural' granularity but in general it depends, at least to some extent, on the way the algorithm is implemented. The communication requirements of the system must be

carefully analysed and matched to the target hardware. Some algorithms will require low latency communication that may suggest a bus based shared memory system is favourable (provided the bus contention is low). Other algorithms may demand high bandwidth but be less sensitive to latency, others still will require a mixture of both. In sequential machines, high level languages allow programmers to write code without specific knowledge of the hardware but to write efficient parallel algorithms this knowledge is essential. For maximum performance, either the hardware must be tailored to the algorithm or the algorithm must be written with regard to the hardware architecture.

3.1.6 Data Consistency

Parallel computers introduce the problem of data consistency where individual processes (possibly on different processors) may believe a data value is different to its real value or when multiple processors attempt to set a data value to conflicting values. In distributed systems, data is usually duplicated in a number of processors so local assignments of a value are not sufficient for other processors to acknowledge a change. In most shared memory systems, the system itself is guaranteed to be consistent but the software must still take care that *it* remains consistent. Usually this consistency involves ensuring that only one processor may ever write to a data item at any one time. For example, if data is written to the same location simultaneously by more than one processor in a shared-memory system the machine will correctly store the two values in which ever order they obtain the bus but the programs themselves have no way of knowing which value is stored after the two writes have been performed.

Data consistency and the associated problem of truth maintenance in knowledge-based

systems is a complex problem but can be avoided if mutual exclusion is forced for accesses to vulnerable data.

3.1.7 Mutual Exclusion, Deadlock and Starvation

In any system there will be some resources, such as shared data, disk drives and I/O cards, that can only be used by one process at a time. To ensure this mutual exclusion is obeyed, in a multitasking system as well as a parallel one, requires some mechanism for locking resources. There are a number of mechanisms for guaranteeing mutual exclusion such as semaphores[1] [43] in shared memory systems and token passing in distributed memory systems. Whichever scheme is used, mutual exclusion brings with it a number of complications and puts extra demands on the programmer. The two fundamental problems, deadlock and starvation must be avoided and in addition to data consistency, constitute the major difficulties in developing parallel applications. Although the two phenomena are simple to explain, identifying them in practice for large, complex systems is very difficult.

Consider a system comprising two processes and two resources that must each be accessed by only one process at a time. Also assume that each process must obtain *both* of these resources simultaneously at some stage in their execution. A naïve approach will simply guard each resource with a semaphore, for example, and ensure that both of the resources are locked before proceeding. However, if both processes attempt to obtain the locks together and they both obtain one of the resources, they will wait indefinitely for the other resource to become free. This state is known as deadlock and is characterised by a number of processes unable to do useful work because they

[1] Semaphores are explained later in Chapter 6.

are blocked by another process in the deadlock group. There is a simple solution to the example given but in more complex cases, potential deadlocks are very difficult to identify.

Whereas a deadlocked system is doing no useful work, a system exhibiting starvation will continue to run to some extent. Starvation occurs when one or more processes in the system are never allowed access to a mutually exclusive resource, although other processes may be given access to it. Starvation is particularly difficult to predict because the programmer must consider the worst case scenario in which groups of processes 'conspire' to deprive another process of a resource.

3.2 Real-Time Systems

The performance of most computer systems is measured in terms of logical correctness. A system must produce the correct answer each time it is run and although it is always desirable to have this answer as soon as possible the answer does not depend on the calculation time. A program that performs some complex calculation may be unusable if it requires a week to compute the answer but the answer is still correct when it does appear. Real-time systems differ from this paradigm because the validity of their output is a function of time and these systems must incorporate not only logical correctness, but also temporal correctness.

There are numerous examples of real-time systems, most of which are used for controlling some physical system. In each case, the real-time controller must respond to the changing environment in which it is working. The controller of an automatic guided vehicle, for example, must be able to read its position sensors that measure how far it

is from a wall, along with its speed, and stop or turn the vehicle before it crashes. The time available to make this decision is dependent on the vehicle speed when the sensors are read but the control system would clearly be incorrect if it allowed the vehicle to crash.

Real-time systems differ widely and range from dedicated microcontrollers to large distributed computer networks. Many real-time systems, including those of interest to this work, involve a number of processes running on some form of computer system. There are two distinct types of real-time process, those that must be run periodically and those that must be run once or in response to certain system conditions. Periodic real-time tasks are described by their period, T, whereas aperiodic real-time tasks are described in terms of their deadline, d. In general, both classes of tasks also have a start time, s and a computation time, c.

3.2.1 Hard and Soft Real-Time

The class of processes with real-time constraints can be divided into two sub-classes, *hard* real-time and *soft* real-time based on the severity of deadline constraints. There is no definitive definition of these terms so they are often used in a different sense than that implied here. For clarity, the definition used in this thesis is as follows.

Hard Real-Time The correctness of these tasks depends on not only the logical correctness but also the timing correctness. A hard real-time task performs no useful function after its deadline. As soon as the deadline is passed, a hard real-time task can be terminated.

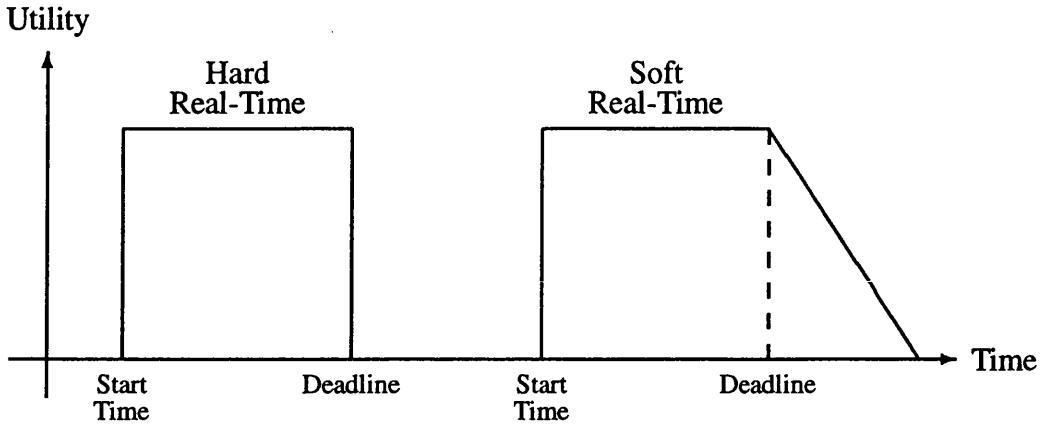

Figure 3.1: Real-Time Utility-Time Graphs

Soft Real-Time Response time is still important for soft real-time tasks but their results are still useful beyond their deadlines. A soft real-time system will still continue to operate correctly if a deadline is occasionally missed. In most soft real-time systems the value (utility) of a task drops after its deadline according to some function of time.

The distinction between soft and hard real-time tasks involves their relative value after the task's deadline. This can be shown clearly on *time-utility* graphs as shown in Figure 3.1. The soft real-time system remains useful after its deadline (although with reduced utility) whereas if the hard real-time task is late, it is totally worthless.

The notion of timing correctness is distinct from time critical. Systems can be soft real-time and time critical or hard real-time and not very time critical. The classification of some systems can be quite difficult as some notion of timeliness may be more appropriate than hard or soft real-time. It is usually clear how these should operate however so the classification is somewhat pointless.

3.3 Scheduling

Scheduling is fundamental to today's computer systems. Most modern operating systems provide a mechanism for *multitasking* enabling a number of processes to run, apparently concurrently, on a single processor. The CPU time is divided amongst the processes and the operating system ensures that each process can behave as if it were the sole user of the computer's resources. In practice, multiprocessor computers also use multitasking on each individual processor because the number of processes will in general exceed the number of processors. The decisions about when processes will be allowed to run and for how long are made by the *scheduler* according to an algorithm that will give the system the desired performance.

Multitasking greatly improves system throughput and efficiency in most cases because it can utilise CPU time that would otherwise be idle. In most computer systems, processes have to interface with slow devices such as disk drives, terminals, printers etc. that are unable to keep pace with the CPU. In single tasking systems, a process that is reading information from disk must make a request and wait for the data to be returned, in which time the CPU is doing no useful work. A multitasking system however can block a process after it sends the disk request and schedule another process in its place. When the disk access returns, typically signalled by an interrupt, the original process can be scheduled once again thereby keeping the processor active at all times. Large multi-user systems can be built following this procedure based on the assumption that most users will be running interactive processes and hence the CPU load for each user will be low.

Multitasking brings with it a number of problems however. Firstly some mechanism for

process swapping must be implemented and secondly a scheduler, which will decide when to swap a process out and which process to swap back in will be needed. Both of these activities will require time and hence will become a system overhead. That is, they will take a percentage of the system resources that could otherwise be used to run one of the current processes. It is important therefore that this overhead is kept as small as possible.

3.4 Real-Time Scheduling

Real-time systems must pay great attention to the scheduler as the overall temporal behaviour of the system, and hence the system correctness, is eventually determined by the scheduling algorithm. In a way the scheduler is the bridge between the future (the execution plan) and the past (the history trace of execution) [44]. The prime consideration of any real-time system is to provide an adequate response to changes in the environment [45] and the scheduling algorithm is central to achieving this. Whereas a multi-user system will strive to implement a 'fair' scheduler, a real-time system requires an algorithm biased towards tasks of higher priority or with tight timing constraints. In a real-time system the scheduling problem becomes one of ensuring that all processes are allowed to run and complete before their deadlines and to be able to degrade gracefully in times of system overload. In practice real-time scheduling algorithms for systems of any reasonable size are highly complex and the development of appropriate scheduling algorithms has been isolated as one of the crucial challenges for the next generation of real-time systems [46].

3.4.1 Static Scheduling

Many real-time systems can be predicted in advance and hence analysed and scheduled off-line. Consider a simple controller that must set the positions of a set of actuators according to the values of a set of input sensors. The controller transfer function will be a differential equation and with a predictable processor, the time required by the control process is determinable in advance and will remain independent of the input data. Even in more complex scenarios it is often possible to predict the run times or worst case run times for each process in the system. This allows the designer to determine the behaviour of the system exactly and even if a number of these simple controllers must run and meet different timing constraints the whole system can still be scheduled off-line. This is known as static scheduling and has the advantage that the system can be guaranteed before installation and as the scheduling process is off-line, more complex scheduling algorithms can be used. It will be seen that even static scheduling of well defined task sets becomes computationally intractable in multiprocessor systems or when a number of mutually exclusive resources are required.

For static scheduling, the start times, execution times and deadlines of all the processes must be known *a priori* and the schedule is calculated in advance. The scheduler in the system is then programmed to perform the schedule as calculated. Such a scheduler will have a low overhead and guaranteed success provided that the *a priori* knowledge was correct. The actual method for calculating the schedule will be similar to those of on-line schedulers, although the removal of time constraints in the calculation of the schedule may allow more optimised solutions to be reached.

3.4.2 Dynamic Scheduling

Unfortunately, not all real-time systems can be analysed off-line. For these systems a dynamic scheduler must be used that has to cope with a number of extra problems. Tasks may arrive unpredictably, system overloads must be managed, resources allocated etc. and all with the added problem of imposing minimum overhead as the dynamic scheduler must actually consume some of the CPU resource it is attempting to control.

Problems that require dynamic scheduling do so because the CPU load of the system is in some way dependent on the input data. This dependency may result in unpredictable computation times or processes may be terminated or spawned depending on input conditions. The dynamic scheduler is able to assess the current status and optimise the system performance at run-time. Consider a system that controls a large chemical process. Under normal circumstances the control requirement may simply be to implement a digital version of a conventional control system. If however a fault occurs, other processes may have to be created to deal with alarms, system shutdown, operator notification, fire extinguishers or evacuation plans. The exact behaviour depends on precisely how bad the fault is and it is at best very difficult, and usually impossible, to predict these conditions in advance. To deal with these situations requires an operating system that is able to make decisions dynamically to try to ensure the correct behaviour.

3.5 Single Processor Scheduling

Scheduling tasks on a single processor is well understood. Work on these algorithms began after industrialisation to schedule work in factories. The problem of allocating

work to workers in order to meet delivery dates and precedence constraints is clearly analogous to the problem of real-time process scheduling.

It was shown by Liu and Layland [47] that for a set of independent periodic tasks running on a single processor, the rate monotonic scheduler is optimal.[2] The rate monotonic scheduler is a simple priority scheduler in that it will always schedule the process with the highest priority provided it is able to run.

The priority of tasks in the rate monotonic scheduler are assigned according to their period. Tasks with smaller periods are considered more important and therefore are given a higher priority. As such,

$$P_i = \frac{1}{T_i} = f_i$$

where P is the process priority, T is its period and f its frequency. The optimality of the rate monotonic scheme relies on the ability to schedule preemptively. Preemptive schedulers allow running tasks to be descheduled before they have completed and continued at a later time, allowing the rate monotonic scheduler to run the highest priority active process at all times and be able to swap processes as higher priority ones become active rather than after completion.

The simplicity of the rate monotonic scheme has allowed it to be analysed extensively. The processor usage given as the proportion of CPU used over time is clearly given by $u = \sum_{i=1}^{n} c_i \cdot f_i$ which must always remain less than or equal to unity. In their paper, Liu

[2] An optimal scheduler will produce a valid schedule (with no timing violations) for any set of processes that can be scheduled by *any* other algorithm.

and Layland derive a least upper bound of processor utilisation which is given by

$$u = \sum_{i=1}^{n} c_i \cdot f_i \leq n(2^{\frac{1}{n}} - 1)$$

where n is the number of processes. This gives a quick schedulability check because any task set whose utilisation is below this value will be schedulable. In practice, this lower bound is approached only when the periods are relative prime whereas utilisations of 100% are possible for harmonic periods. In general this bound is very pessimistic so keeping the utilisation below this level is very inefficient.

The rate monotonic scheduler in its standard form only works for well-defined periodic independent task sets. The algorithm has been widely extended to include aperiodic tasks but these systems remain biased towards periodic tasks. One such extension is to use a periodic server that is given all the spare CPU time available. The aperiodic tasks then 'apply' for time from this server.

Other optimal schedulers exist for single processor systems. The earliest deadline and the minimum laxity algorithms have both been proved optimal [48]. The earliest deadline is a fixed priority scheme like the rate monotonic scheduler but the priority given is inversely proportional to the task deadline. Tasks that have earlier deadlines therefore have higher priorities. The earliest deadline scheduler does not require all the tasks to be periodic but it can handle periodic tasks by considering them as tasks with a deadline of T. When each instance of the task completes, a new one is created with a deadline of $2T$. The earliest deadline gives good processor utilisation and has the significant advantage of being independent of the computation time, making it a useful scheduling solution for systems where c_i is unknown.

The minimum laxity scheduler is a dynamic priority scheme in which the highest

priority is given to the process with the lowest laxity. The laxity, l_i of a task is defined as the time available before that task must be scheduled to meet its deadline and is therefore given by

$$l_i = d_i - c_i' - t$$

where c_i' is the remaining computation time of the task (ie. c_i minus the CPU time already used). Clearly, if l_i is negative it cannot be scheduled before its deadline so in the case of hard real-time systems the task can be discarded. The minimum laxity scheme is dynamic in that the laxity of every task except the one currently running decreases (and hence the priority increases) with time.

Scheduler optimality depends on being able to schedule a schedulable task set. All the above algorithms are optimal in that sense but each behaves very differently under overload conditions. The rate monotonic scheduler has no regard for deadlines explicitly and the earliest deadline scheduler, because it does not use the value of c_i, cannot detect a failure until the task's deadline. The minimum laxity algorithm is able to use the c_i information to reject tasks earlier (when the laxity becomes negative) and hence prevent the CPU being used to run a task that will fail. The behaviour of a scheduling algorithm under overload is known as its *stability* where more stable algorithms behave better under overload. The earliest deadline scheduler can therefore be classed as optimal but unstable. All these schemes however ignore the real importance of each task. The problem is the use of a single priority value which must reflect the often contradictory demands of both timing constraints and importance.

In most real-time systems, some tasks will be more important than others and as such should be allowed to run in preference to lower priority tasks in times of system overload. Sha *et al.* [49] derived a priority transform for incorporating real priorities

into the rate monotonic scheme. Their transform works by dividing more important processes up into a number of smaller (and hence more important in the eyes of the scheduler) blocks. In general however, a set of prioritised real-time tasks becomes much more difficult to schedule.

When the computation time is not known, the task of the scheduler becomes exceptionally difficult to perform under overload conditions. In practice, the computation times of many tasks are not known accurately but it is usually better to give the scheduler as much information as possible upon which to deduce its schedule. The computation time of a single task may vary widely depending on the input data so a compromise has to be made between accuracy and efficiency. If the worst case execution time is used, as must be the case in very critical systems, a task may be rejected that could actually have run because it would have taken far less than its stated computation time. Similarly, if mean computation times are used, CPU time could be given to tasks that have no chance of completing. In soft real-time systems average computation times can ensure an acceptable processor utilisation at the cost of missing some deadlines. In practice this compromise depends heavily on the application and the available task information.

The algorithms discussed have all relied on independent processes, with no inter-process communication, synchronisation or mutually exclusive resources. If any of these are considered, the scheduling problem quickly becomes intractable. Mok [50] has shown that scheduling any system with processes using mutually exclusive resources locked by semaphores is NP-hard[3] and goes on to say that in fact *most* problems with both resource and time constraints are also NP-hard. To solve the scheduling problem

[3]For a definition and description of NP-completeness, see Garey and Johnson [51].

under these conditions requires either justification for ignoring the dependency effect or implementing a sub-optimal heuristic algorithm.

3.6 Multiprocessor Scheduling

Dertouzos and Mok [48] showed that the optimal schedulers for single processor systems do not remain optimal in multiprocessor cases. A number of polynomial algorithms have been derived for simple, static scenarios such as the $O(n^3)$ algorithm by Horn [52] for mapping independent tasks to identical processors. In fact, the earliest deadline algorithm remains optimal if all tasks have unity computation time but these scenarios bare little relation to realistic problems. Mok and Dertouzos went on to prove that there can be no optimal algorithm for scheduling tasks on more than one processor if all the start-times are not known *a priori* even if there are no restrictions on preemption and no precedence or mutual exclusion constraints. Given that optimal schedules cannot be found, the task of any multiprocessor scheduler must be to provide an adequate, sub-optimal solution.

Static scheduling on multiprocessor systems has been investigated widely and many sub-optimal algorithms have been derived that employ heuristic rules to limit the problem search space to a manageable size. Typically these algorithms are intended to minimise the run-time of an application rather than meet any real-time criteria, and use various measures to indicate 'good' schedules. Various systems by Houstis [53], Sarje and Sagar [54], Chu and Lan [55] and others use measures of inter-processor communication, precedence constraints, load balance, and total execution times in their search heuristics. Many of these have produced near optimal results but do

require prior knowledge of all system tasks and with the advantage of running off-line can take a considerable time to run.

Dynamic scheduling on multiprocessors, especially for real-time systems, is an extremely complex problem. To date, few solutions have been proposed for these schedulers but it does seem clear that centralised systems, where a master processor is performing the scheduling task and farming out the tasks to its worker processors will be unable to cope with the demands of a real-time environment, especially in overload conditions.

Some of the most promising work, being developed in different forms by a number of groups, uses some form a distributed global scheduler working above the local schedulers that are present on each processor. This approach allows a known optimal algorithm to be used locally and some form of heuristics to control the distribution of tasks globally. Smith's Contract Net [56], work by Alijani and Wedde [57] and the work by Stankovic and his colleagues at the University of Massachusetts [58–60] all use this type of architecture to good effect.

In the work by Stankovic *et al.*, newly created tasks will arrive at some node where the local scheduler will either accept the task and guarantee to meet its deadline, or reject it and attempt to find another processor capable of executing the process. The local guarantee routine used is a complex algorithm [61], involving both precedence and resource constraints so a dedicated processor is added to each node to remove the scheduling overhead. The distribution policy used by the global scheduler is a hybrid one, combining focussed addressing and bidding. In focussed addressing, as used in the Contract Net, each node stores information about the load levels of every other node and sends the new process to the one with the highest surplus. At regular intervals

each node must notify the others of their present state. A bidding system relies on sending out messages to other nodes describing the task, who return bids reflecting their ability to schedule this task. The task is then sent to the highest bidder. Clearly, focussed addressing is based on out-of-date information whereas bidding incurs a high communication overhead so Stankovic *et al.* have developed a hybrid system. Their work has been going for some time and more recently the system has been extended to include system reliability [62] and work is underway to incorporate the ideas developed into a real-time kernel called Spring [63]. Most of this work has only been simulated so far and a great deal of work remains to be done.

Scheduler stability is another major problem area for dynamic distributed schedulers. Stankovic [64] in a paper concerning stability directly, concludes that the problem is extremely difficult to resolve due to the typically large number of 'tunable' parameters present in any heuristic approach. The notion of stability is subjective and specific to each algorithm and operating environment so there must be some automatic way of setting the stability parameters automatically if the issue is ever to be analysed. One potential solution to this may lie with a knowledge based system to set up the scheduling algorithm and even to tune the system during operation.

3.7 OS support for scheduling

Many real-time systems are implemented on purpose built hardware using in-house real-time kernels rather than on commercial platforms. The reason for this is the lack of support given to real-time processes in most operating systems, particularly the lack of real-time scheduling. For this project, aiming to develop a widely usable and portable

framework, the use of a commercially supported operating system is essential. In the computer system world at present, Unix remains the most widely used multithreaded operating system so this must clearly be *a* target platform although the system should remain as independent as possible from the operating system. The development system was built under the Helios operating system (see Section 4.2.1) which performs no scheduling itself but relies on the transputer's built in hardware scheduler which uses a round-robin scheme. The portable and efficient scheduler designed for this research is described in Section 5.4.1 and requires very few services from the host operating system. For clarity, a brief description of the Unix and transputer schedulers are given below, followed by a short description of the Posix initiative.

3.7.1 The Unix Scheduler

The Unix scheduling scheme, implemented by the kernel, was designed to provide a responsive system in multi-user environments and is based on *multi-level feedback queues*. Each process has a priority level and the kernel maintains a list of priority levels, each containing a list of the active processes of that level. The next process to run is chosen as the first process on the highest priority non-empty queue and processes of equal priority are scheduled in a round-robin manner. Once scheduled, a process may run until it terminates, blocks on some event, uses up its time-slice (typically about 100ms) or until a higher priority process becomes available and preempts it.

The problem for real-time systems is the mechanism for the assignment of priorities. The priorities are not under the control of the programmer, instead the system adjusts the priorities dynamically according to the CPU usage of the process. At each time

slice, the kernel updates process priorities according to the formula

$$p\_user = PUSER + \left[\frac{p\_cpu}{4}\right] + 2 \cdot p\_nice$$

where a lower *p_user* denotes a higher priority, *p_cpu* is a measure of recent CPU usage and the *p_nice* term is user-definable to give an offset to the calculated priority. This scheme ensures that CPU bound processes are reduced in priority while processes waiting on events have their priorities increased. As such, interactive processes waiting on user input retain high priority due to their low CPU usage and hence the system remains responsive.

The 'fair-share' scheduling strategy, described by Thompson [65] as having a "desirable negative feedback characteristic" is disastrous for real-time systems as users have no direct control over the process priorities. The scheduling scheme described later recognises this fact and does not rely on the Unix scheduler.

Many attempts have been made to add real-time capabilities to the Unix operating system. Mert [66], an example of such an extension, provides control over process scheduling parameters and memory residency but as with the other developments, the ideas have not found their way into the Unix distribution. Current evolutions, spurred by the Posix initiative and resulting in the release of Unix System V release 4 do include some limited real-time support but most systems in use today are still using earlier versions.

3.7.2 The Transputer Scheduler

The transputer was designed primarily to be a component in parallel machines. It is intended to efficiently support the CSP paradigm and as such provides a hardware scheduler that allows processes to be managed very cheaply with context switches taking less than 1μs. The scheduler controls two process queues, one for high priority tasks and one for all other tasks. High priority processes are always scheduled whenever they become runnable and are allowed to run to completion (or until they are blocked). Low priority processes are round-robin scheduled with a time-slice of about 10ms. Preemption occurs only when a low priority task uses a complete time slice or if a low priority process is running when a high priority process becomes runnable. As the transputer provides such an efficient scheduler in hardware and this is uncontrollable by software, Helios does not consider the scheduling problem at all. Some kernel processes run at high priority (as this ensures atomicity) and all user processes are run on the low priority queue.

For most software, the transputer scheduler is perfectly adequate as usually it is only necessary to know the processes will run eventually and to allow a number of processes to run interleaved on one processor. If a process is particularly important, it can simply be executed on its own processor. Unfortunately, for any practical, real-time system where the number of processors are limited and specific deadlines have to be met, this scheduling algorithm is inadequate. It is simply impossible to have the control over process scheduling that is necessary in such systems. Therefore, as was the case for Unix, a scheduling mechanism that does not rely on the host scheduler is required.

There have been a number of attempts at developing real-time schedulers and real-time

kernels on the transputer, perhaps the most mature of which is the TRANS-RTXc kernel [67]. All these systems achieve real-time behaviour by residing on the processor's high priority queue and explicitly modifying the processor's low-priority queue so the required process will always be selected. The technique is obviously very system specific so is unsuitable for a portable solution and actually does not perform much better than the portable solution proposed later in this thesis.

3.7.3 The Posix standard

Since 1988, the IEEE has been working on the Portable Operating System Interface for Computer Environments (Posix) standard. The standard aims to define the operating system interface as seen by the programmer to ease portability and further the open systems initiatives. The original Posix document, IEEE P1003.1 [68] specified a complete set of system calls with specified arguments and ranges and their behaviour. This standard did not explicitly mention real-time.

Since then, the Posix committee has been expanded to 16 working groups, three of which are concerned with real-time issues. P1003.4 is dedicated to real-time extensions, P1003.4a deals with thread (light weight processes) extensions and P1003.13 defines application environment profiles for real-time application support. These three groups have produced the real-time Posix extensions known collectively as Posix.4. To date only a small number of operating systems claim to be Posix.4 compliant but the standard does provide the extensions required, such as priority scheduling, to build real-time systems and as such is likely to be incorporated as standard in future operating systems.

The Posix standard defines the operating system interface in detail but fails to address

the problem of a standard memory model. Although Posix will directly support uniprocessors and shared-memory parallel machines, there is no support for scalable parallel processing architectures—a serious omission that will need to be resolved in the future.

3.8 Other Scheduling Techniques

It has been shown that the scheduling problem, even for relatively simple cases, is very difficult to solve and its solution becomes impossible to optimise for more complex systems. For this reason, there has been some interest recently in applying new techniques to the scheduling problem. A good summary of knowledge based scheduling approaches can be found in the paper Noronha and Sarma [69] but a brief overview of three such methods is given below.

3.8.1 Scheduling with Expert Systems

There is great scope for applying expert system technology to the scheduling process. It has been shown that many algorithms are difficult to 'tune' for particular applications and this may be a potential use of expert systems. Thesen *et al.* [70] used a simple system scheme to switch between scheduling algorithms depending on the system state and produced some impressive results. More complex systems will need considerable research but may well provide the flexibility required to produce quality schedules under wide variations in system condition.

3.8.2 Scheduling with Neural Networks

Neural network technology is being applied to a wide variety of problems. The technology uses simple neuron structures that are linked together to form a network. Each neuron receives a number of inputs which are weighted and summed to produce an output. Networks of these simple neurons are capable of performing complex pattern recognition tasks when the individual weights can be set appropriately. The elegance of neural networks comes from the fact that these weights can be learnt by the network itself during a 'training' phase. The network is subjected to a set of representative inputs along with the expected outputs and by repeated application, the network can adjust its weights to recognise the training set. Once trained, provided the training set was representative and covered the entire problem space, the network will be able to solve problems that were not in the original set.

For some scheduling problems, where the necessary training data can be derived, neural networks seem to be a promising possibility. Johnston and Adorf [71] have developed a long term real-time scheduler that is in use for scheduling observations on the Hubble Space Telescope. The system deals with time constraints, precedence and uncertainty and schedules both soft and hard real-time processes. The complexity of this scheduling problem has led the team to derive a new neural network architecture to produce their very encouraging results. It seems clear that this technology still has a long development time ahead of it before it becomes widely applicable to scheduling problems.

3.8.3 Scheduling with Genetic Algorithms

Genetic algorithms are a recent invention pioneered by John Holland at the University of Michigan [72]. They are essentially search algorithms that mimic the natural processes of evolution to perform their search.

Genetic algorithms operate on a set, or population, of *chromosomes* that each, in some way, encode a solution to a given problem. Each individual chromosome is evaluated by a routine that is able to assess its quality and these "fitness" values are used to influence the next generation of individuals. In exactly the same way as natural evolution, fitter individuals contribute more to the next generation than the less able. In the reproduction process the children of the next generation inherit characteristics from each of their parents, possibly combining the best features from each and becoming a better individual than either of the parents. With these two operations of parent selection biased by fitness and the randomised crossover of chromosome information to the child, a population will converge naturally to some 'ideal' individual. This ideal may however be sub-optimal because the reproduction process essentially concentrates the good ideas already present in the population. To introduce diversity and new ideas, a genetic algorithm uses another operation based on natural evolution—mutation. At each reproduction stage, a small number of individuals will, at random, be mutated typically by inverting a single bit in the chromosome.

The genetic algorithm itself is independent of the problem. It relies only on being able to evaluate each chromosome so that this value can be used to bias future reproduction. The algorithm knows nothing about how the chromosomes relate to the real problem and nothing about the characteristics of the problem itself. The genetic algorithm merely

continues to adapt the population in such a way that the best individuals are investigated and merged into new, hopefully better solutions. Of course, many reproductions will produce children that are very bad solutions but these will die off naturally in later generations.

A detailed description of genetic algorithms is beyond the scope of this thesis. A good introduction can be found in Davis [73] and a more rigourous analysis is given in Goldberg [74]. Goldberg describes the mathematical foundation of genetic algorithms and shows the important distinction between a randomised search and a random search.

The scheduling problem can be described as finding the optimal ordering of a set of tasks with resource and timing constraints. A genetic algorithm is well suited to this type of problem and many researches have applied them to this field. One example, by Syswerda [75], uses a genetic algorithm to schedule usage of the System Integration Test Station (SITS) laboratory of the U.S. Navy. This laboratory contains numerous facilities and is made available to developers of F-14 fighter jets. Multiple users can use the laboratory simultaneously provided they do not require the same resources. The problem is a classic resource constraint problem, complicated by the addition of job priority, emergency work, cancellations, equipment breakdown etc.

Syswerda has produced some good results from this work and his system has been shown to be very close to optimal in that it will find a schedule if one exists in the vast majority of cases. His solution however takes a considerable time to run and requires a large number of iterations. At the present time, this precludes the use of genetic algorithms in dynamic schedulers but as faster processors are developed and the genetic algorithm techniques are refined, it may be possible to utilise this powerful technology in the future.

3.9 Summary

This chapter has covered two main areas—parallel systems and real-time systems. Firstly parallel processing was introduced and some of the problems of implementing parallel systems were highlighted. Real-time systems were then outlined before discussing the particular issue of real-time scheduling. Scheduling in both single processor and multiprocessor systems was discussed and the work of others in this field was reviewed. The chapter ended with a brief review of some of the new technologies that are being applied to the scheduling problem.

Chapter 4

Computing Facilities and System Software

This chapter describes the computing facilities used during the course of this work. The main implementation and development environment was an Inmos T800 transputer based parallel computer running an operating system called Helios. Both the hardware and the system software will be described in some detail. The University of Bath parallel diesel engine simulator used in the course of this work will also be discussed.

4.1 Hardware

Large scale simulators, including the real-time diesel engine simulator and a project to model the electro-mechanical behaviour of the UK National Grid in real-time have been developed at Bath for a number of years. These simulators require vast computing power to achieve their real-time objectives and commercial systems capable of delivering this have either not existed or been prohibitively expensive. To produce the required processing power, it was decided to utilise parallel processing techniques and to develop an in-house parallel computer based around cheap, widely available micro-processors. The first parallel machines, built in the early 1980's, were based on the Motorola 68000 [76–78]. These computers were superseded by 68020 based systems employing a similar system architecture [79]. The most recent system, based

on the T800 transputer, uses a completely new architecture and was developed by Dunn *et al.* [80].

4.1.1 The T800 Transputer

The Inmos transputer family is a range of products specifically designed to be used in parallel computers. In any parallel system, processors must be able to communicate with each other if they are to work together. The transputer range of processors was, at the time, unique in providing on-chip communication hardware known as *links*. Links provide synchronised bidirectional serial communication between processors and with the outside world at speeds of up to 20 MBits/second. Each link comprises an input and an output channel and devices are connected simply by connecting the input channel of one device to the output channel of the other device and vice versa.

The T800 [81] was the most powerful member of the transputer range available at the time and was chosen because of the computing power it delivered at a relatively cheap unit cost. The T800 is a 32 bit processor with an integral hardware 64 bit floating point unit that Inmos claims will perform 10 MIPS (Million Instructions Per Second) and a sustained 1.5 MFLOPS (Million FLoating-point Operations Per Second) from a 20 MHz device. The single substrate also contains 4 KBytes of fast internal DRAM and four independent link interfaces. The four link interfaces enable arrays of processors to be created with a variety of interconnection patterns. A block diagram of the T800 transputer is shown in Figure 4.1.

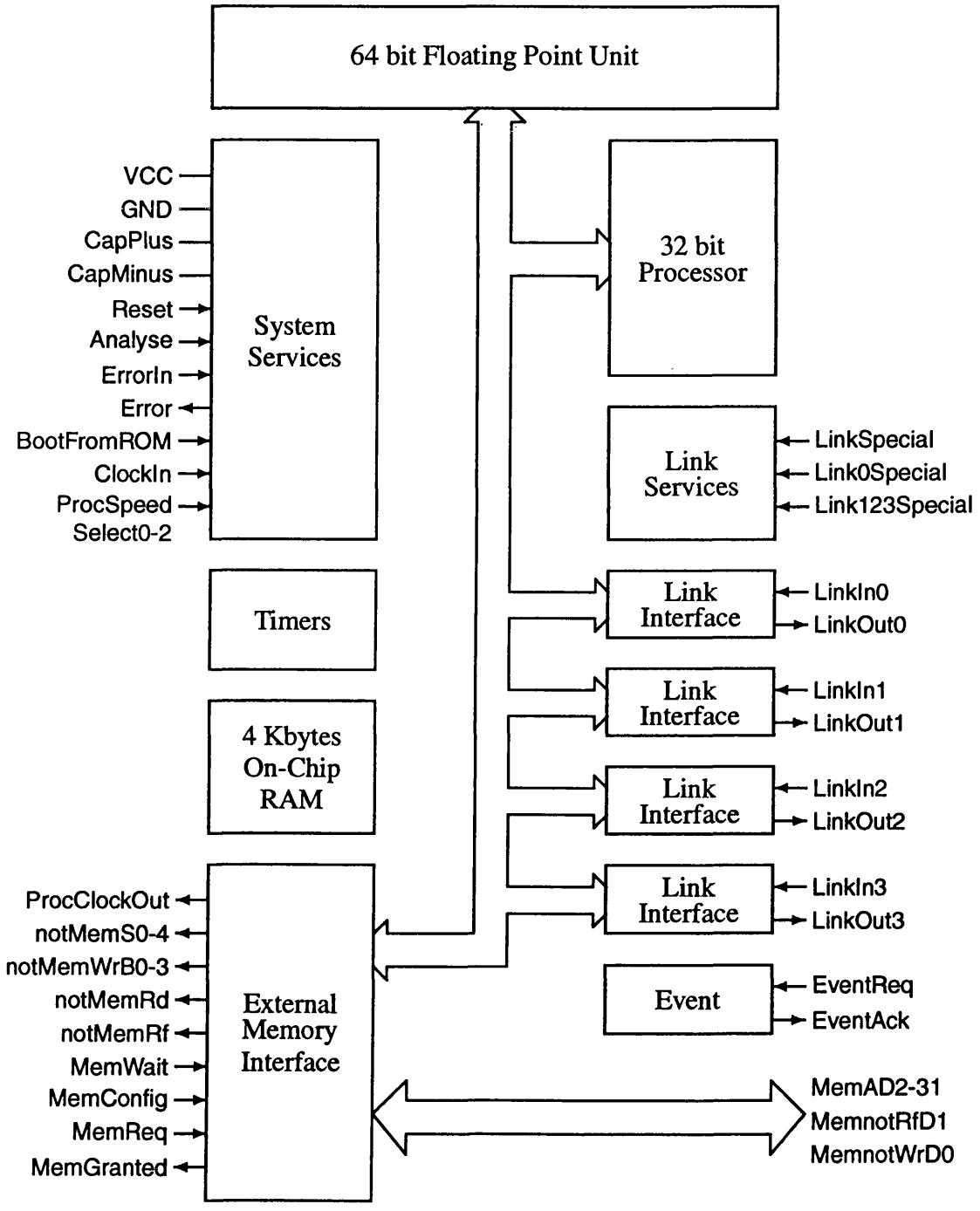

Figure 4.1: IMS T800 Block Diagram

4.1.2 CSP, `occam` and the Transputer

Tony Hoare (Oxford University) and David May (Inmos) designed a language called `occam` [82] that supports the CSP programming model. `occam` allows processes to be created simply and provides channels over which they may communicate with each other and the outside world. Inmos developed the transputer primarily as an `occam` engine, intended to efficiently execute programs structured as groups of communicating processes. The transputer itself can be viewed at a process level as a micro-processor 'process' with four channels with which to communicate with other transputers. `occam` programs will typically result in large numbers of processes and these can be distributed over a network of transputers as required. Two processes communicate in exactly the same way whether they are on the same processor or they are located on different processors and are communicating via a transputer link. Efficient execution of `occam` programs requires efficient communication and light weight processes. In general there will be many processes running on each processor and these must be interleaved efficiently. Processes are therefore supported in hardware by the transputer making their creation and handling very efficient. The transputer contains a hardware scheduler that employs a simple time-sliced round-robin scheduling scheme. Internally, there are just 6, 32 bit registers but the transputer will only ever deschedule processes at certain *descheduling instructions*. By controlling the points at which scheduling occurs it is possible to reduce the processor state that needs to be saved at each context-switch so the transputer is able to switch between processes in less than one micro-second.

The transputer is also useful in its own right as a general purpose processor. Unfortunately however, it lacks some facilities expected of modern processors as these were not required by the intended CSP programming model. The most significant omis-

sion, especially for operating system support, is the lack of any memory management hardware or support for external memory management. Memory management is not required by `occam` programs because processes are unable to access memory outside of their own domain—there is no pointer type. For general purpose processors however this is a serious problem as it is impossible to guarantee that a rogue process will not overwrite the entire memory and this makes supporting an operating system and debugging code very difficult. As mentioned in the previous chapter, the hardware scheduler provided by the transputer is also unsuitable for real-time systems where it is essential to maintain control over the process scheduler.

4.1.3 The T800 Parallel Computer

The specification for the new parallel computer was determined by the requirements of the diesel engine simulator as this was the most computationally demanding of the applications. The diesel engine simulator has a minimum time step of only 80 micro-seconds when simulating an engine running at 2100 rpm at a 1 degree resolution, in which time a great deal of both calculation and communication must be performed. The problem was analysed assuming a transputer network interconnected by links. This showed that the worst loaded link in the simulator must receive 8, 32 bit words and transmit 8, 32 bit words per step which would, according to the Inmos specification, require about 28 micro-seconds. These figures ignore any synchronisation or message identification so realistically about 50% of the available time step is taken up with communication [83]. This communication cost was considered too high so a new computer architecture was developed that would provide a much higher communication bandwidth.

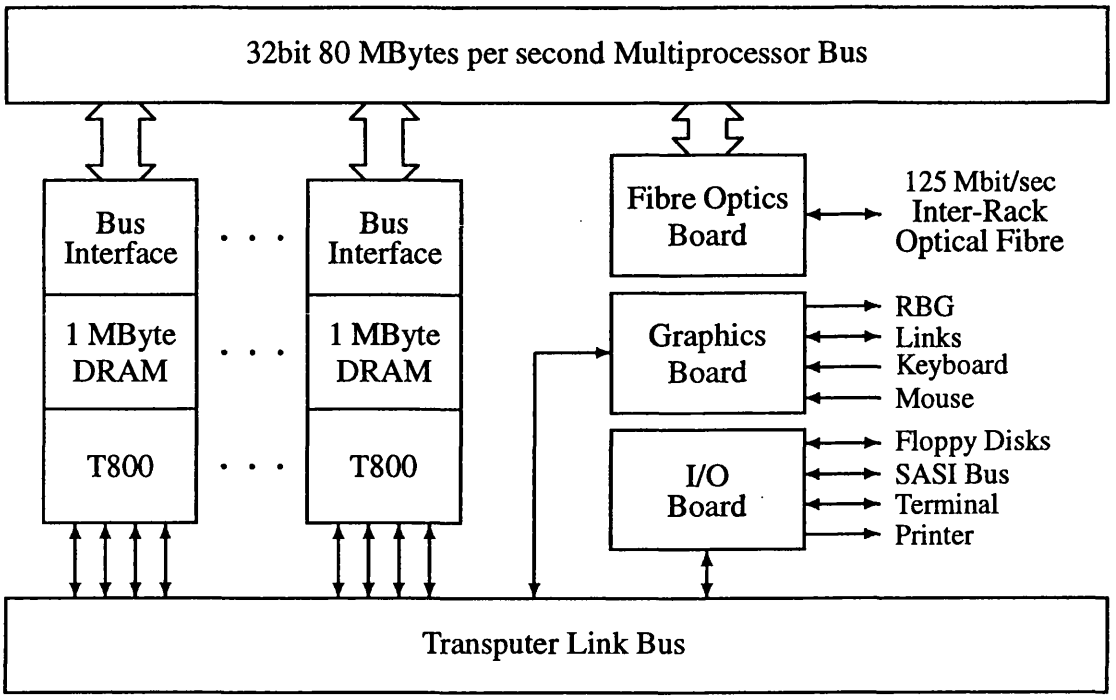

Figure 4.2: The T800 Based Parallel Computer

To achieve the necessary communication speeds, a shared memory architecture was developed as shown in Figure 4.2. Up to 16 processing nodes are connected together via a backplane, along with an I/O board for connection to storage devices, terminal, printer etc., a graphics board that provides a high resolution colour graphics facility and a fibre optic board used for inter-rack communication. The 32 bit backplane bus has an overall bandwidth of 80 MBytes per second. Each rack of 16 processors can communicate with other racks via an optical fibre link, with the racks interconnected in a hierarchical configuration. The architecture provides homogeneous shared-memory throughout the system with 1 MByte of DRAM located on each processing node. With the 4GBytes address space of the T800 this allows up to 4096 transputers (256 racks) to be interconnected giving a total bandwidth of over 20 GBytes/second.

The Processing Nodes

Each processing node contains a T800 transputer with 1 MByte of dynamic RAM. The processors may communicate with each other in two ways, either using the 32 bit shared memory bus or via standard transputer links. The four links of each T800 are routed through the backplane to a link patch area so the system topology can be configured as required. For most purposes a static topology is sufficient but it is possible to insert a software configurable link topology card into this slot. A block diagram of the processing node is shown in Figure 4.3.

The on-board DRAM is effectively dual-ported by the local and external access control logic on each node. The T800 bus is isolated from the RAM with a set of local buffers and from the shared memory backplane bus via the local to external buffers. The local decode determines whether each memory cycle is a local memory access or an external memory access and enables the appropriate arbiter. The arbiters then gain access to the necessary bus and carry out the memory cycle. For external write cycles, the local bus is latched and released so the local processor may continue to operate while the external arbitration and the backplane cycle are carried out. This arrangement allows external writes to be carried out at the same speed as local accesses provided that the previous cycle has completed before the next write access.

Similarly, the external to local buffers will latch any backplane cycles when other boards attempt to address the local memory and complete the cycle when the local bus can be obtained.

The memory map of the parallel computer is shown in Figure 4.4. Each node contains 1 MByte of RAM which is mapped on to the global address map according to its

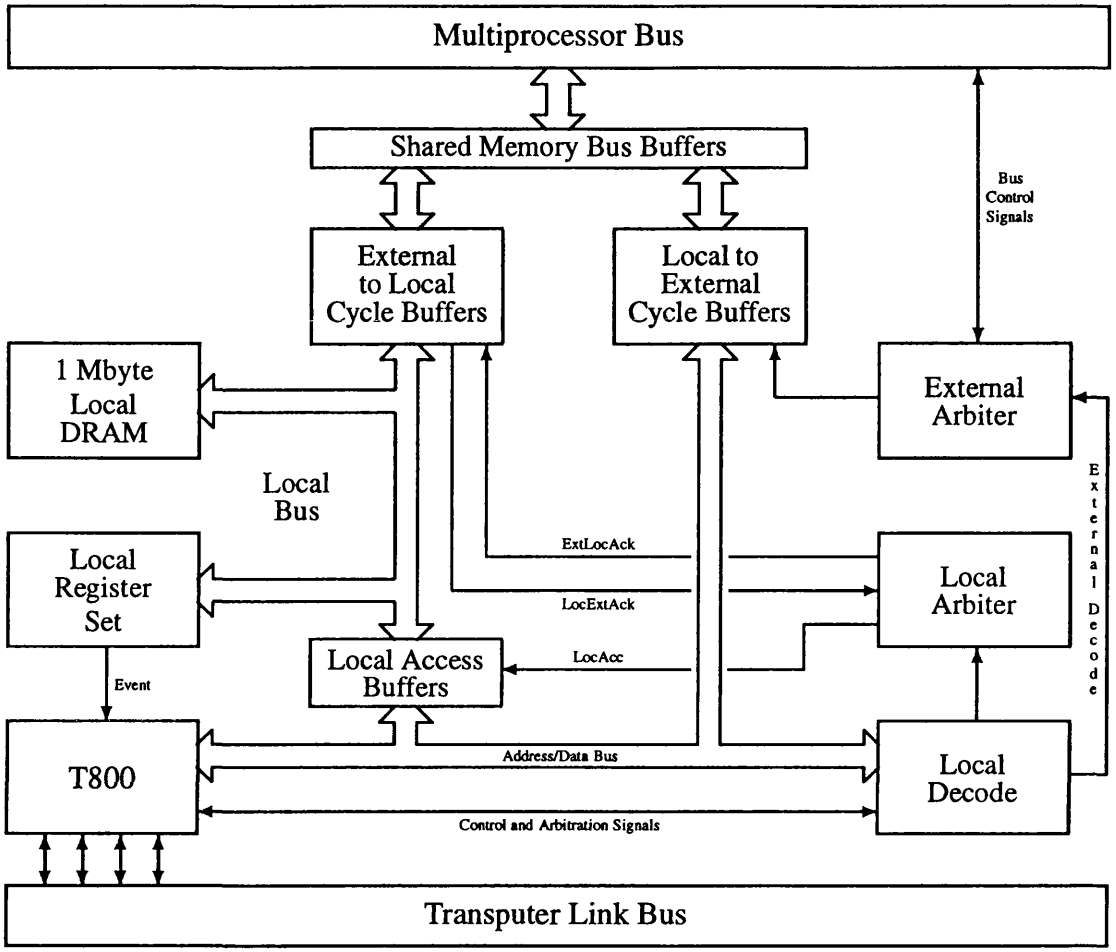

Figure 4.3: A Single Processing Node

rack number and position within that rack. Each rack addresses 16 MBytes of RAM, allowing a maximum of 256 racks to be used within the 32 bit address space. The transputer considers memory to begin at the most negative address (80000000 Hex) and extend to the most positive address (7FFFFFFF Hex) so the local memory must extend from 8000000 to 80100000[1]. The bottom 4 KBytes of the transputers address space is the on-chip RAM and a further 4 KBytes at the top of the local memory is

[1]This actually excludes a rack number 80 so the system is limited to 255 racks (4080 processors) in practice.

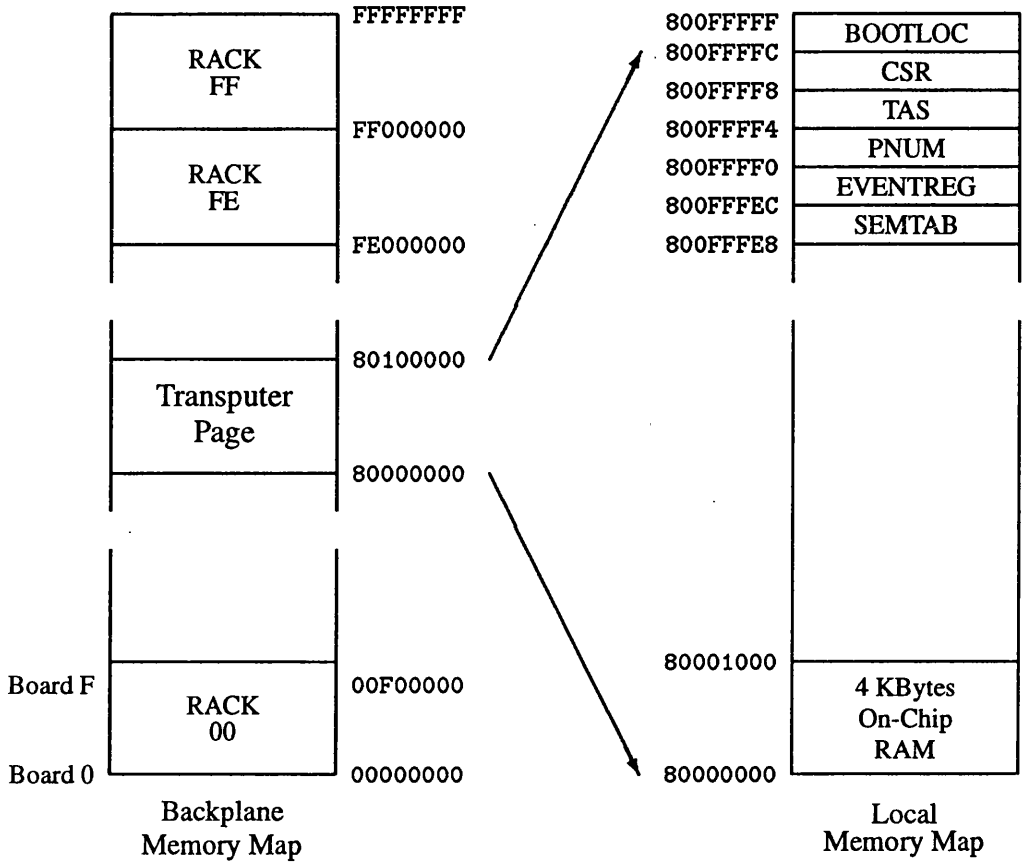

Figure 4.4: The Parallel Computer Memory Map

reserved in this system for special purpose registers.

The Special Purpose Registers

The special purpose registers are shown in Table 4.1 and are mainly concerned with system housekeeping such as the bootstrap location and the node status registers. Two of these registers are of particular interest, the test-and-set register, `TAS`, and the event register, `EVENTREG`.

The `TAS` location provides each board with an indivisible test-and-set facility. Whenever

Address	Name	Description
800FFFFC	BOOTLOC	Boot location
800FFFF8	CSR	Control and Status Register
800FFFF4	TAS	Test and Set
800FFFF0	PNUM	Processor Number
800FFFEC	EVENTREG	Event Register
800FFFE8	SEMTAB	Semaphore Table

Table 4.1: Special Purpose Registers

a read cycle accesses this location, its value is returned and the location is set to logical 1 in one indivisible cycle. This enables the transputers to implement mutual exclusion. The SEMTAB location is provided to point at a semaphore table that is accessed via the test-and-set location, thereby providing any number of effective TAS locations.

The EVENTREG is used to produce a hardware event on the local processor. When a write cycle accesses the location, the write completes and the transputer's event pin is asserted. Normally an event handler will be installed on each processor to deal with these asynchronous interrupts which are cleared by reading the value of the register. Before the event is cleared, all other writes to this location will fail guaranteeing atomicity. In this way, processors may cause events on remote processors and this facility has been used in this project for implementing remote semaphores.

The Backplane

The backplane is designed to connect a maximum of 16 processing nodes in a shared memory architecture. The backplane bus and the bus protocol are not processor dependent so a heterogeneous system could be constructed if required. When memory

cycles are destined off-board, the local state machines latch the cycle and release the local bus. The state machines then perform the backplane cycle while the local processor can continue to operate. Provided that the frequency of external writes is not too high, the local processor can continue to operate at full speed. Similarly, when other nodes access the local memory, the entire operation is performed by local state machines and the processor is not involved.

Each backplane memory cycle is completed in a single cycle of the backplane clock, running nominally at 20 MHz. Backplane write cycles appear on the bus and are latched on the destination processor by decode logic that is monitoring the backplane. For efficiency, inter-processor reads are divided into two phases, an address phase and a data phase. A processor wishing to read an external memory address performs the address phase, a backplane access in which the address bus contains the address of the data required and the data bus contains the local location into which the data should be returned. The address phase will be latched by the processor containing the required data. After some indeterminate time, when the control logic on that board has obtained access to its local bus and retrieved the data value, it initiates the data phase of the cycle writing the data back to the requesting processor. Typically, for a 20 MHz bus, there is about 500 nanoseconds between the address and data phases effectively allowing the communication between 10 pairs of processors to be interleaved.

It will not always be possible for a processing node to accept a backplane cycle. Two signals are present on the backplane to account for this, SUCCESS and FAIL. Each processing node has decode logic that is monitoring the backplane constantly and if it accepts a backplane cycle meant for it, it will assert SUCCESS. If it is unable to accept the cycle at that time, it asserts FAIL. A failure causes the initiating processor

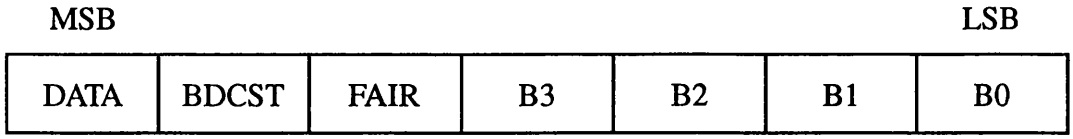

Figure 4.5: Bus Arbitration Priority Scheme

to automatically repeat the cycle 250 nanoseconds later.

Bus Arbitration

To achieve optimum communication bandwidth, an arbitration scheme had to be devised that could operate at the same speed as the bus itself (ie. arbitrating in less than 50 nanoseconds). The arbitration is distributed and performed in parallel with a backplane access, arbitrating for the right to use the backplane on the *next* cycle. A priority scheme, with an in built fairness mechanism is used to break conflicts.

Each board in the rack has a number (ranging from 0 to 15) determined by its location in the rack which forms the lower part of the priority word as shown in Figure 4.5. The upper three bits are used to alter the priority under certain conditions. The DATA bit ensures that the returning data phase of accesses are always given high priority as a processor will be waiting for this data. The FAIR bit is asserted by a board that has failed to gain the bus after 16 time slots and ensures that processors do not get locked out.

Broadcast

The T800 parallel machine has a special backplane cycle known as *broadcast.* This cycle allows a processor to write information to a location on every board in the system in one operation. These broadcast cycles involve every processor so the BDCST bit in the arbitration priority is set to assure they are completed quickly. When a broadcast cycle is performed, each board decodes the access as if it were intended for them and reply with SUCCESS or FAIL signals as before. If any board asserts FAIL, the cycle will be reissued until a completely successful cycle occurs. A processor that has already accepted the data can ignore these retries and continue processing.

4.1.4 The I/O Card

The I/O card occupies another slot on the backplane and provides the system with access to the outside world. A diagram of the I/O card is shown in Figure 4.6. The I/O card is based on the Philips SCC68070 [84], a 68000 compatible processor with memory management, two DMA controllers, a serial port and an inter-integrated circuit (I^2C) interface integrated into the device. The card provides a floppy disk interface, a $SASI^2$ bus for connection to a hard disk and tape streamer, a real-time clock, a parallel centronics port and a terminal connection. The board also contains an IMSC012 [85] link adapter for communication with the transputers. The board was designed as the author's BSc. final year project [86] following work by Hafeez [87].

[2]Shugart Associates Systems Interface.

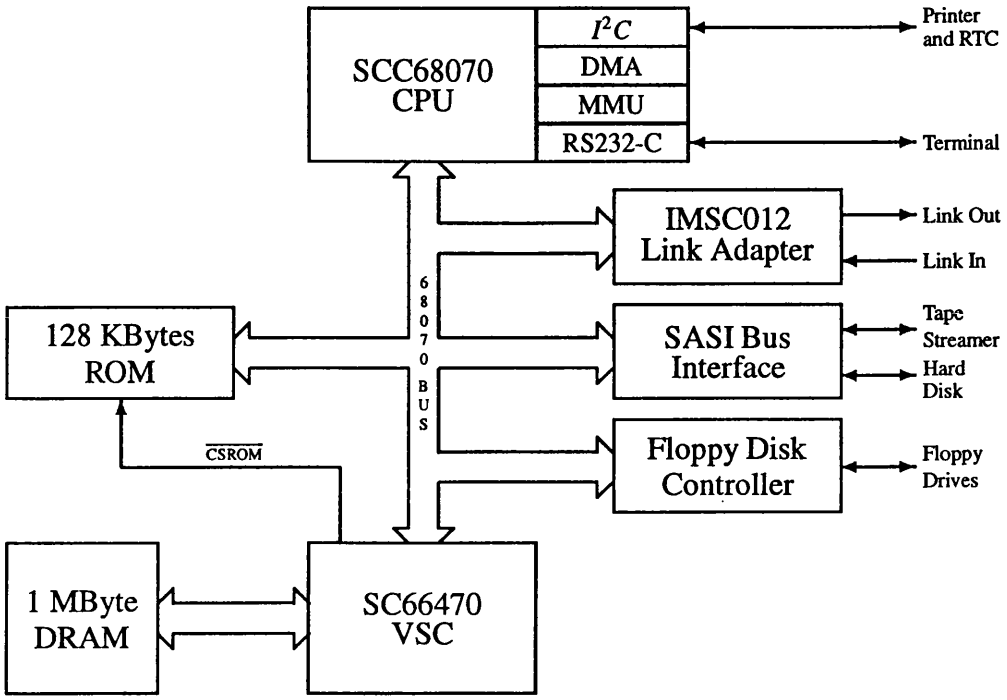

Figure 4.6: The Input/Output Board

4.1.5 The Graphics Card

Another slot in the system rack is occupied by a graphics card to provide the system with high resolution bit-mapped colour graphics. The graphics board contains an SCC66470 Video System Controller (VSC) [88] to produce the graphics which is served by two processors, a 68070 and a T800 as shown in Figure 4.7. The board contains 4 MBytes of RAM for the T800 and another 1 MByte that is shared between the two processors and used as the video RAM. The board also contains a mouse interface and a keyboard interface that are used primarily for X-Window application development.

The VSC is an integrated device that contains a 68000 compatible memory interface, a coprocessor interface and a bit mapped colour graphics controller. The graphics

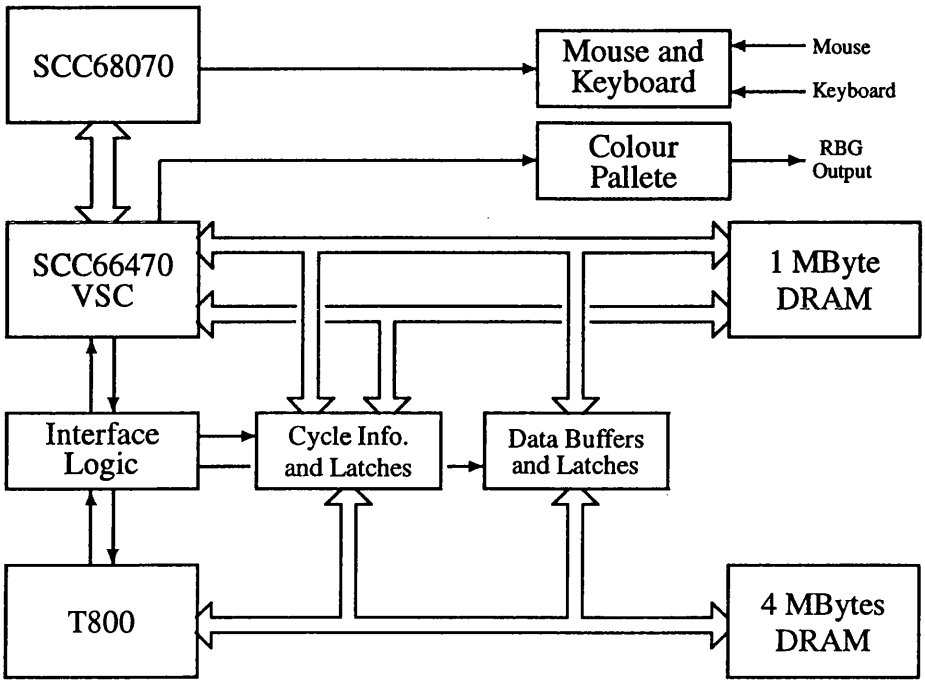

Figure 4.7: The Graphics Card

capability will produce a range of display modes from 320x240 with 256 colours to 768x480 with 16 colours. On the graphics card, an Inmos G178 Colour Palette allows these colours to be chosen from a palette of 16.3 million.

The graphics board also contains the link patch area due to its position in the rack. Every transputer link is routed via this slot and the graphics board can be used as a patch area to select the required system topology. The links of the T800 located on the graphic board are also routed into this area. These machines may be connected to external transputer systems using links driven by differential line drivers. These drivers are also present on the graphics board and allow up to three links to be connected to external devices. These have been used extensively for connecting the parallel computers to IBM PC compatible machines that contain a single T800 plug-in board and thereby sharing the resource amongst a number of users.

4.2 Software

The natural choice of implementation software on a transputer system would be occam but for this project, occam has a number of disadvantages. It is a specialised language for transputers and as such limits the portability of any software produced under it. More fundamentally, occam does not support dynamic memory allocation making its use for dynamic real-time systems limited. In addition, the strict computation model imposed by occam with no concept of a pointer type would make it impossible to support the backplane hardware. The T800 parallel machine was intended to be used for the development of software (as opposed to just running the software that was written and compiled on another machine) and as such an operating system was essential. The operating system chosen was Helios [89].

4.2.1 The Helios Operating System

The Helios operating system was designed primarily for transputer systems although it is now supported on a wide range of processors. The majority of Helios is written in the high-level language C [90], relying only on a small system specific kernel making the operating system relatively easy to port to new processors. Helios is a distributed operating system built loosely within the CSP framework. Helios includes networking software for booting and managing arbitrary networks of possibly different processors and for distributing tasks across them in a semi-automatic way. Processes under Helios communicate by message-passing, the natural choice for transputer based systems, that is supported at a higher level than the raw transputer links. Helios processes communicate directly and the messages are routed, possibly via other processors, to

the destination process automatically.

As with all operating systems, Helios controls the resources of the machine. It provides a consistent means of accessing these using a client-server model. Each resource is controlled by a server and accesses are made through this using the General Server Protocol (GSP). Helios servers are stateless in that they do not retain information about previous requests to them. This inevitably causes some overhead due to the increased message traffic but it ensures a design goal of Helios to be fault-tolerant. If a processor crashes under Helios it is rebooted and reconnected into the network automatically and because the servers are stateless, nothing is lost.

The Helios programming environment has been designed to be similar to a Unix [91, 92] system and includes a shell based on C-Shell. Software (source) compatibility with Unix based systems has been an aim of Helios since its inception. The current goal is to become as compatible as possible with the two established Unix standards, AT&T's System V release 4 [93] and BSD 4.3 [94] as well as the proposed Posix standard (IEEE 1003.1-1988) [68]. This has already been achieved to a some extent with the provision of Posix and BSD libraries, allowing many programs to be ported to Helios as easily as they could be ported to another Unix platform.

Helios supports a number of high-level programming languages. In addition to the ANSI Standard C [95] compiler supplied as standard, Helios also supports a FORTRAN 77 compiler and C++. The C++ support is based on AT&T's `cfront` package that translates C++ into C and uses the system's own C compiler to produce the executable programs.

A Helios system consists of a nucleus running on every processor in the network, a

collection of servers running on those processors, shared libraries loaded to certain processors and a collection of user applications. Helios utilises shared libraries to reduce the considerable overhead caused by linking libraries to applications (both in time and memory).

4.2.2 The Helios Nucleus

Each processor in a Helios network runs a copy of the *nucleus*. The nucleus is a collection of system services and shared libraries that control the resources of each processor. The nucleus provides the necessary hooks for the higher level networking software that manages the collection of processors as a whole. The nucleus provides the user with an abstraction of the underlying system that is identical regardless of the actual target hardware. It is only the nucleus, or more specifically the kernel, that requires explicit knowledge of the hardware. The Helios nucleus, shown in Figure 4.8, consists of six separate modules: the kernel; the utility library; the server library; the system library; the loader and the processor manager.

The kernel is the lowest level of the nucleus and manages the hardware resources of the processor. It implements the message passing and routing that lies at the heart of the Helios system. In addition the kernel controls dynamic memory allocation, event multiplexing, semaphore operations and other hardware specific tasks. It is the kernel that provides the greatest challenge in porting Helios to new processors.

The libraries provide the applications, networking software, and all higher level processes access to the kernel routines via a uniform interface. The system library includes most of the system routines required to implement Helios clients and in particular, de-

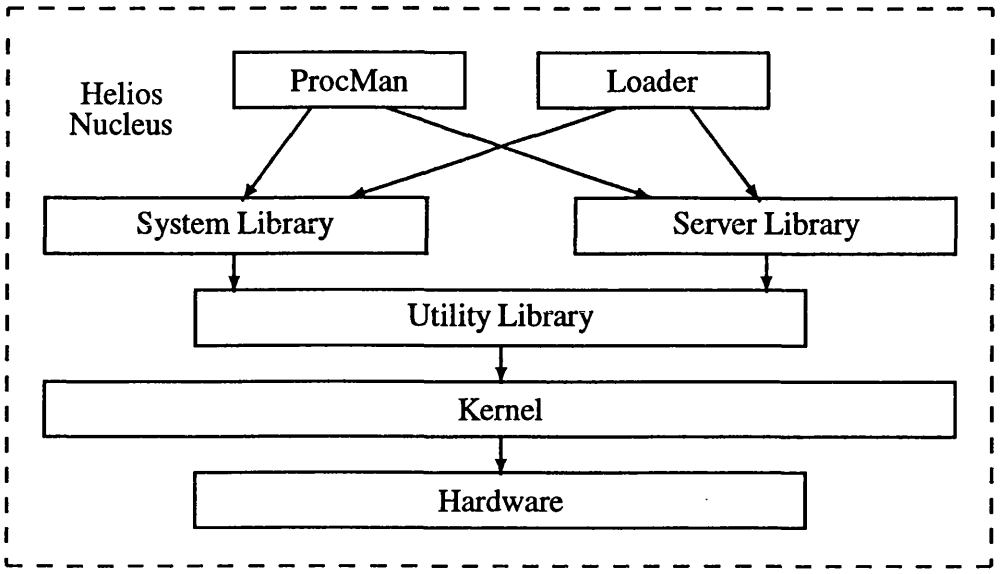

Figure 4.8: The Structure of the Helios Nucleus

fines the General Server Protocol. The server library provides additional routines for the creation of new servers while the utility library allows access to a number of general kernel services that are of use to application processes.

The loader server manages all the code loaded on to the processor. It loads and unloads both code and resident modules and enables processes to share common libraries. The use of resident modules, such as the C library, allows processes to share a common copy of the routines rather than duplicating the code. This reduces process load times as well as memory usage.

The processor manager controls the tasks[3] running on each processor. It is responsible for creating new tasks, managing them while they are running and dismantling them

[3]Helios defines a *task* as a program entity in the state of execution that contains at least one, and probably several, concurrent *processes* sharing the environment of the task. These terms correspond to the Unix definitions of *processes* and *threads*. Unless this distinction is necessary, the terms task and process will be used interchangeably.

when they terminate. Specifically, it controls the signal mechanism for each task and ensures that any resources allocated to a task are released when it eventually terminates. For each task, the processor manager creates an *I/O Controller* (IOC) process to act as an intermediary between the task and the system. The IOC's are responsible for locating the named objects (servers etc.) that tasks wish to use and do so either from the local name table or by conducting a distributed search to neighbouring processors and eventually across the entire network.

4.3 Real-Time Diesel Engine Simulator

A parallel real-time diesel engine simulator was also used during the course of this work. The simulator was developed at the University of Bath over a number of years, and this section will describe its development and current status.

The work on diesel engine simulation has been driven by the goal of creating a real-time engine model, that is a model that can simulate a diesel engine and produce results at the same rate as a real diesel engine. Clearly, real-time becomes easier to achieve if the model used is less accurate but this work aimed to produce a useful real-time tool and hence an accurate model had to be used.

The first generation diesel engine simulator was developed by Jones [3] who saw the need for parallel processing to be used to achieve the desired simulation speeds. His model was based on a serial FORTRAN simulation package called SPICE[4] written by Charlton [96]. Jones's version was written in BCPL and ran on a 68000 based shared memory parallel computer. The simulator uses the filling and emptying model in which

[4] Simulation Package for Internal Combustion Engines.

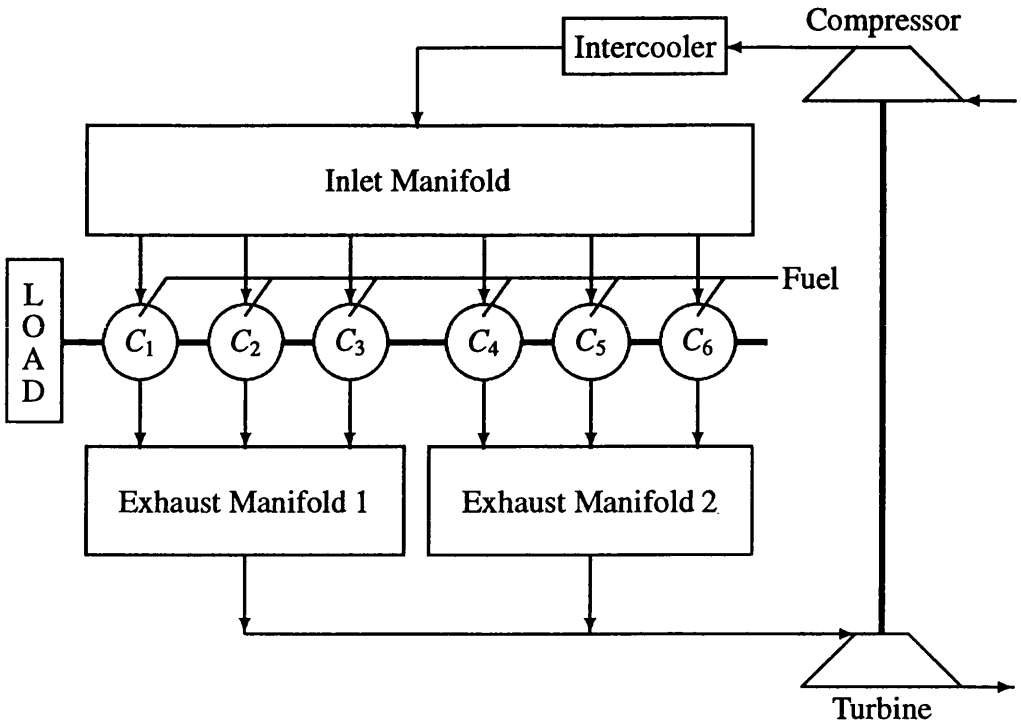

Figure 4.9: Leyland TL11 Diesel Engine Schematic

the engine is represented as a set of interconnected thermodynamic control volumes, along with the dynamics and control functions that describe the physical engine.

To aid the verification of the simulator, it was decided to model an 11 litre Leyland TL11 truck engine as a fully instrumented engine of this type is installed in the School of Mechanical Engineering, University of Bath. The engine is a six-cylinder turbo-charged diesel engine, a block diagram of which is shown in Figure 4.9.

The parallelism exploited in the simulator is known as *geometric parallelism*, in which the distribution of the tasks is based on the physical distribution of the system. In this case, each control volume is calculated on a separate processor and other processors are used to calculate the engine dynamics, control actuators and numerical stability. The structure of the simulator is shown in Figure 4.10 and clearly reflects the physical

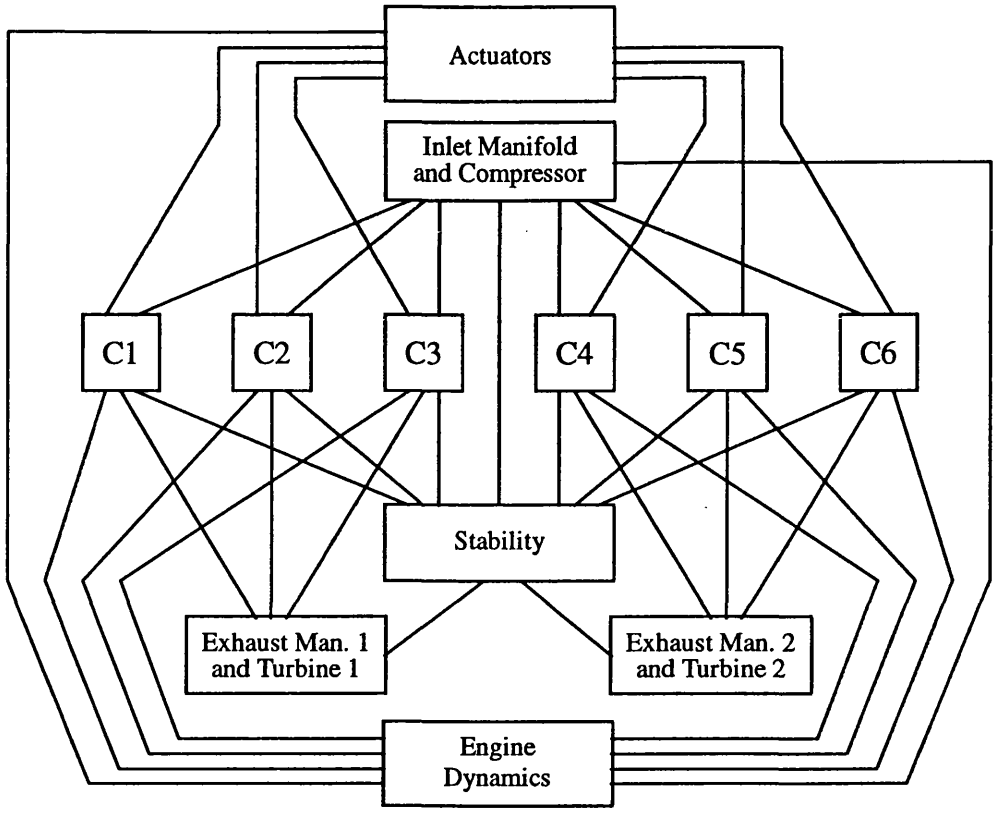

Figure 4.10: The Task Organisation of the Parallel Diesel Engine Simulator, showing the inter-task communication paths

structure of the engine.

This model was further developed by Haysom [4] who utilised a faster, Motorola 68020 based, parallel machine and a new algorithm to greatly improve the model performance. The latest development was undertaken by Shamail [2] who translated the model from BCPL to C and ported it to the new T800 based computer under Helios. He also generalised the model by removing hard coded engine parameters from the simulator and using a start up file to define the engine configuration. This version has since been updated by the author to include a graphics interface and re-implemented as a Helios server to allow other programs to communicate with the simulator.

The simulator now runs on 13 T800 transputers using the high-speed 32 bit backplane for communication. Typically, the simulator runs at about 1 / 15th of real-time—that is the simulator can simulate about 100 revolutions per minute of an engine running at 1500 rpm. Although the real-time goal has not been achieved the model is very fast compared to similar serial simulators. The simulation algorithm is now highly developed and there appears little to be gained by working to improve the algorithm further. The T800 is now outdated in terms of processing power. The Intel i860 performs a factor of 10 better than the T800 and faster processors are being released. It would take considerable effort (if possible at all) to improve the simulator algorithms by a factor of 2. Already, by using modern processors a 10 fold speedup could be obtained and clearly it will not be long before faster processors make the real-time simulation of diesel engines feasible.

4.4 Summary

This chapter has described the computing facilities used throughout this work. Some background information was given about the transputer itself before describing the transputer based shared memory parallel machine in detail. The machine is used with the Helios operating system and this was also described in some detail. The chapter ended with a summary of previous work on a parallel diesel engine simulator that was used in the application of this work to diesel engine fault diagnosis.

Chapter 5
The Design of Grape

This chapter describes the design of the parallel real-time knowledge-based system known as Grape. The system architecture is discussed first, followed by more detailed descriptions of the individual system elements. At each stage the reasons for particular design decisions will be outlined. The implementation details have been omitted from this discussion to help clarify the design issues and are presented in the following chapter.

5.1 System Architecture

It was shown in Chapter 2 that parallelism has been exploited in knowledge-based systems in a number of ways. Solutions vary from the instruction level parallelism used to implement a parallel inference engine, through distributed systems that partition rules (or rule clauses) across a set of processors, to sets of co-operating but independent inference engines. The choice of this granularity is important to the resulting behaviour and performance of the system and depends to some extent on the hardware platform and the application. To provide the required real-time response and the ability to respond to external stimuli and changing conditions, a system of co-operating independent inference engines was chosen because of the added functionality that this architecture provides even on a single processor.

The Grape architecture allows a number of inference engines to run simultaneously, each executing their own rule base and referencing both private and shared fact bases. The diagnostic problem can be solved well using this type of system using a hierarchical diagnostic model in which new specialised processes are created as the system focusses in on the fault. Having a parallel system gives added advantage by allowing a number of possible fault paths to be followed simultaneously or a number of workers to co-operate on a single problem. A hierarchical diagnostic model is also useful where a response time has to be guaranteed. Under these conditions, the lowest level inference engines that have produced results represent the system's best response so far. The system architecture is shown in Figure 5.1.

Grape is controlled at the lowest level by its scheduler, which is responsible for the low level execution and control of each individual process as well as the implementation of the local and global scheduling schemes. If an external event needs to be dealt with, the scheduler, through its use of the task priorities, can immediately schedule the appropriate task or create a new task specifically. The use of a dynamic scheduler produces a more responsive system. The scheduler is divided into two modules, the *agenda manager* and the *dispatcher*. The agenda manager is responsible for analysing the timing and priority constraints of the current process pool and creating an agenda of future execution that allows the process constraints to be met as optimally as possible. In a single processor version, processes which cannot be added successfully to the agenda are rejected (hard real-time tasks) whereas in multiprocessor systems the agenda manager goes on to migrate the task to a suitable processor. The dispatcher provides the low level process control and is responsible for implementing the agenda produced by the agenda manager. The dispatcher responds to requests for new processes to be initiated and schedules the agenda manager itself when a new agenda is required.

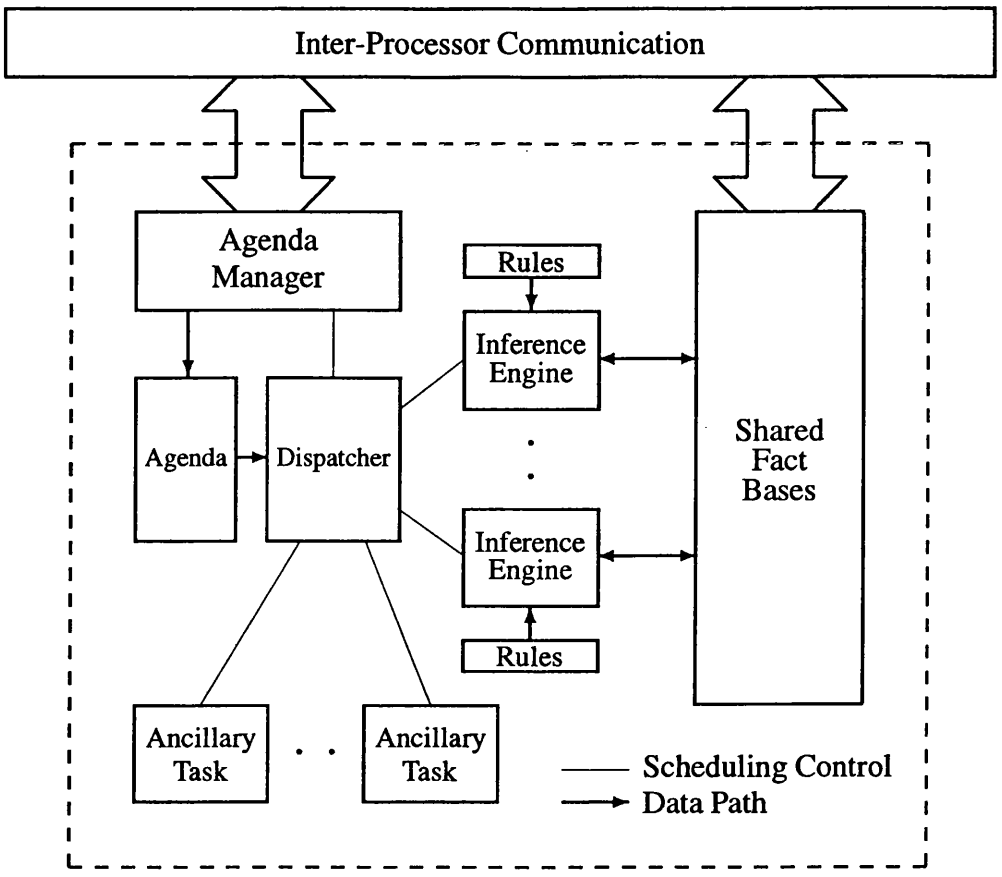

Figure 5.1: The Grape Architecture Block Diagram

The main processing elements in the system are the inference engines which apply the knowledge, encoded by the domain expert and knowledge engineer, to the diagnostic problem. Other auxiliary processes are also controlled by the dispatcher, such as the data acquisition system and any state monitoring processes. The priority values of tasks can be changed at run-time either by the task itself or by some other task. Providing the system with dynamic priority values allows the reasoning process to the directed by some global policy under the control of a high priority task.

5.2 The Inference Engine

The inference engine is rule-based. A rule-based model of knowledge representation was chosen because of its conceptual simplicity and its ability to represent the diagnostic process in a natural manner. The knowledge used to build a diagnostic system is likely to be expressed by the domain expert as a set of '`if..., then...`' statements. The ability to see the inference trace of the rule firings is useful to explain the system's reasoning to the user and is also of great help in the development and debugging of the rule bases.

For the parallel real-time capabilities, a number of attributes are required of the inference engine. It must be able to co-exist with multiple instances of itself, share data in a secure way and be able to spawn new tasks as a result of certain conditions. To benefit fully from a system of multiple co-operating inference engines each one must be relatively small, have a high inference rate and be lightweight in terms of both time and system resources. These conditions do not allow a state-saving inferencing algorithm to be used as the cost of generating the data-flow graph would be too high. The paradigm encourages small rule bases which again reduces or even overturns the benefit to be gained from state-saving.

Each inference engine is an instance of the same process and as such cannot have its rules and fact bases compiled into the code. In some systems, coding the knowledge directly into a standard language, such as C and compiling a dedicated inference engine is an acceptable solution. In general however, it is ill-advised to use hard-coded knowledge because of the problems of updating and changing the rule base.

In the Grape system, each inference engine accesses a single rule base (to which it has sole access) which is specified on its command line. The rules are written in a high level rule language and are compiled into the rule bases used by the inference engines. Each rule base will access a number of possibly shared fact bases which themselves are written in another dedicated language. The fact base language allows facts to be arranged into object structures and for the the types and default values of each fact to be defined. As with the rule bases, the source fact bases are compiled into the form used by the inference engine.

Each rule base includes references to the fact bases used by the rules within it and the inference engine uses this information to link up with the relevant fact bases. A central table of fact bases is maintained that is used to gain access to them. Each fact base is uniquely named and this name is hashed to locate its entry in a central reference table. When the inference engine starts, it uses this table to determine if the relevant fact bases are already present in memory. If so, it simply links to them and increments the user count associated with the fact base. If the fact base is not loaded, then the inference engine will load it and make the base public by entering its details in the fact base table. For efficiency, fact bases can be preloaded on system start-up or cached. Normally, when an inference engine exits and decouples from the fact bases, it will detect if the user count for a particular base is zero and unload it. If a fact base is designated cached, it will be loaded by the first inference engine that requires it but it will not be unloaded. As the fact base table is a shared resource it must be accessed atomically and a semaphore is provided for this purpose.

Under Helios it is also possible to cache code using a facility provided by the kernel. Copies of the inference engine and other frequently used tasks can be stored on each

processor allowing faster process spawning and task migration. As Helios allows code to be shared, only one copy of the code is actually required on each processor reducing overheads and making more efficient use of the system memory.

The fact base structure reduces the amount of dynamic memory management that needs to be performed at run-time which in other systems can cause very severe performance limitations. The structure and type of each object and each fact are known at compile time allowing the inference engine to allocate a single area of memory directly. Individual instances of facts may still be created at run-time and these are held in linked lists but the majority of the data can be referenced directly. When new inference engines are started, the rule base is loaded and each rule clause is stored along with the explicit addresses of the fact base objects to which it refers. This linking process involves some searching of the fact base at load time but greatly increases the execution speed as no further searching is required.

5.2.1 The Knowledge Representation Language

The knowledge consists of definitions for both the rule and fact bases. For ease of use, maintainability and hence reduced errors the knowledge, as programmed by the user, should be represented in a simple but comprehensive language. In this system, the rule and fact bases are *compiled* into the form required by the inference engine off-line. This approach has a number of advantages:

- The rule and fact bases are known to be syntactically and semantically correct before execution. This means that the inference engine does not have to perform time consuming validation, it can read in the knowledge knowing exactly what

format it will be in and that it will be able to execute these rules (it of course does not know if the rules will perform as intended as the compiler cannot validate the functionality of the knowledge!).

- Compiled knowledge is in a much more compact form than the source knowledge. This allows the user to express the knowledge in a clear way, commenting the rule base and ensuring readability and maintainability without affecting the system performance. The reduced load time of smaller fact and rule bases helps to minimise the disk I/O overhead when inference engines are first executed.

- Some of the interpretation of the rules is performed by the compiler reducing the work required at run-time. Also, clauses are ordered by the rule base compiler to allow maximum efficiency in the inferencing process. Variable instantiation is costly in rule clauses so ordering clauses by increasing number of variables can produce significant improvements. It is clearly inefficient to satisfy a number of clauses containing variables only to find out that a simple comparison clause fails.

- In a real-time environment especially, it is always advisable to perform as much validation as possible off-line.

The rule and fact bases are written using two languages designed for Grape and are compiled with dedicated compilers. The structure of the language is C like, although the syntax and semantics are very different. The Backus-Naur Forms (BNF) [97] of these language grammars are included in Appendix A. Chapter 8 describes the use of these languages in the development of a diesel engine diagnostic system.

The fact bases are specified as a collection of object definitions. Each object is in turn a collection of fact definitions. The arrangement of facts into objects allows complex physical entities to be clearly represented. A `cylinder` object for example may include facts such as `temperature`, `pressure`, `volume` etc. Arranging the data in this form allows a more natural representation and hence eases the development process. Individual facts will normally hold a single value (along with a time-stamp, odds, recency number etc.), but may be declared as a 'trend' variable in which case specified number of past values are stored. Trend variables allow the application to reason about the past performance of the system. By default 10 trend values will be stored but alternative levels may be defined. Trend values can be referenced directly in rules.

Each fact in the fact base is *typed*, which allows the compilers to perform type checking at compile time. In this way, all run-time type checking can be avoided and this clearly helps to improve efficiency and reduce programming errors. Each fact definition may also include a default value that will be used in the absence of more specific data. Along with the definition of objects in the fact base, a number of static instances may also be defined. The advantage of this facility is that these instances can be accessed directly at run-time (using the rule base linking mentioned earlier) without any indirection. Dynamically created instances always incur the slight penalty of at least one level of indirection.

The rules language specifies a set of rules, each with one or more condition clauses and one or more action clauses. Condition clauses are made up from the comparison of two expressions, one of which will (in general) contain a reference to a fact base object. The objects may be either specific objects or variables that will be instantiated

to all objects of a particular type. Any number of variables may be used in a rule and may be shared between clauses. Each distinct variable must have a unique binding that satisfies every clause in which it is used if the rule is to be satisfied. The action clauses can include assignments, operations on the instantiated variables or one of a number of commands. These commands are used to interact with the user, spawn new processes, change process priorities and add and remove dynamic object instances.

5.2.2 The Matching Phase

The matching phase is relatively simple and is very fast. The rule clauses are pre-ordered by the compiler in order of an increasing number of variables for the reasons mentioned on Page 96. Each rule is evaluated in turn by testing to see if each clause can be satisfied. In simple clauses this will involve a comparison operation that will either succeed or fail. When variables are included, the variable is instantiated with all possible objects that satisfy the particular condition. Each variable instantiation must be consistent for every clause in the rule. Whenever a rule is completely satisfied, it is eligible to fire. The conflict resolution stage is carried out concurrently with the matching phase by remembering only the most eligible firable rule.

5.2.3 Conflict Resolution

Conflict resolution can greatly affect the system performance but there is no single solution as to how it should be performed. For this reason, a number of algorithms are included and may be selected by the developer. The two main schemes are rule priority in which each rule is given a priority value and the highest priority firable rule is chosen

and recency. Resolving conflicts by recency involves finding the rule which contains the newest or most recently changed data. The scheme is particularly useful because it favours recent events and therefore increases the responsiveness of the system. Two recency protocols have been implemented, one that uses the single most recent piece of data in the condition clauses and another that uses the sum of the recencies of all the data in the clauses. In general, a rule firing does not prevent that rule from being satisfied again so unless the consequence of the rule is to enable other, higher priority (or more eligible) rules, the rule will be reselected again during the next cycle of the inference engine. This is usually undesirable as the inference process is making no progress so by default a rule will not fire twice on the same data. If this behaviour is required, it can be requested.

5.2.4 Uncertainty

The uncertainty scheme chosen for the development system is similar to that used in the MYCIN system. This was selected because a mathematical analysis of the engine diagnostic system would have been impossible and the MYCIN approach appears to work well in this type of environment. For future development, the uncertainty has been implemented in an abstract way allowing other algorithms to be implemented very easily. For example, it may be that some statistical data will become available in the future about diesel engine faults (or whatever other field the Grape system is applied to) and this may allow a Bayesian approach to be used.

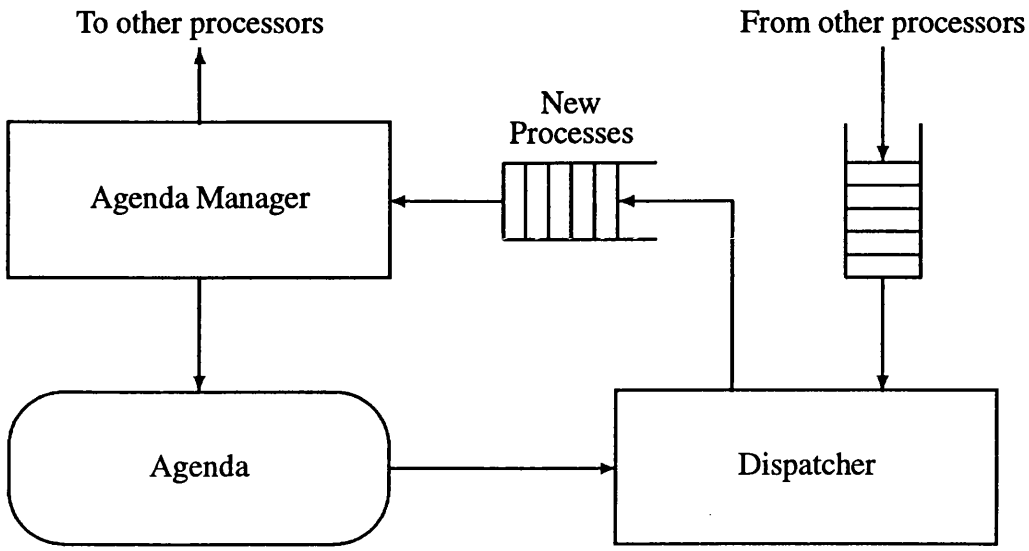

Figure 5.2: Scheduler Queue Architecture

5.3 The Dispatcher

The dispatcher maintains a set of process descriptors, one for each process under its control, which contain the timing constraints, priority and other accounting details of the process. The list of current process descriptors is passed to the agenda manager to construct the agenda which is, in effect, an ordering of the current process set.

As the agenda is ordered in terms of *scheduling priority*, a combination of both timing constraints and task priorities, the dispatcher needs only to ensure that the highest priority process that has passed its start-time and that is free to run is given access to the CPU. The dispatcher starts from the head of the list of running processes and schedules the first process that has passed its start-time. When that process returns the CPU, the dispatcher returns to the head of the agenda and again schedules the first eligible process. This scheme is extended to allow for processes of equal to be round-robin scheduled. The dispatcher achieves this by moving the process descriptor

of the returning process to after all processes of equal priority before returning to the head of the list to find the next process to schedule.

The agenda manager itself is also under the control of the dispatcher and is scheduled whenever a new agenda is needed. The agenda will become invalid whenever a new process is created or the priority of a process is changed but the agenda manager is not necessarily scheduled immediately because this can lead to a subtle form of priority inversion. Consider the case where a high priority task is running and spawns a low priority task to perform some non-time critical function. If the dispatcher responds to this new process request by creating a new process descriptor, loading the task and scheduling the agenda manager, then the high priority process will have been effectively preempted by the low priority task. This problem is solved with the addition of an agenda level variable, `agendaLevel`. The agenda level variable is set to the maximum of its current value and the priority of the newly created task (or the priority to which a process is changed) and the dispatcher only ever schedules the agenda manager when the agenda level is higher than the priority of the process that it would otherwise select. While higher priority processes remain active, the requests for new processes are queued awaiting the eventual invocation of the agenda manager.

The dispatcher is also responsible for certain system housekeeping functions. The computation times of tasks for example need to reflect the computation time still required and hence need to be updated each time a process is run. Timing information is also maintained by the dispatcher for system performance results and the shared fact base memory on each processor is also controlled by the dispatcher.

As already mentioned, the Unix and Helios schedulers cannot be used to provide real-time. Instead a co-operative scheme, described in Section 6.3.1 has been designed that

allows the dispatcher to maintain control over the usage of the CPU and implement a real-time scheduling policy. The use of a co-operative scheme necessitates some minor modifications to the application programs in order that they interact correctly with the dispatcher but these modifications are very simple.

5.4 The Agenda Manager

The agenda manager is called to produce a new agenda based on the current process list. The dispatcher will call the agenda manager when a new process arrives or an old process terminates and also when a deadline has been missed (provided the event is of high enough priority). If a deadline has been missed it may be appropriate to kill the "rogue" process but it may also be appropriate to give it extra time so that its deadline is missed by as short a time as possible. This depends on the type of process that has missed its deadline and the agenda manager must take this into account. A process has one of the following types:

- `HARD_REALTIME` if the result of the process is meaningless if the deadline is missed.
- `SOFT_REALTIME` if there is a deadline that should be met but where the result is still useful even if it is late.
- `PERIODIC` if the process must be scheduled every T seconds where T is given in the Process Descriptor.
- `BACKGROUND` if the task has no deadline.

The agenda manager implements an algorithmic scheduling scheme. In the event of a missed deadline, the agenda manager will only kill the process if it has a hard real-time deadline. Otherwise, the agenda will be re-arranged to ensure that the process is completed as soon as possible.

5.4.1 Scheduling Algorithms

The agenda manager must implement a scheduling policy that will ensure an optimal, or acceptably sub-optimal performance. For single processors, it has been shown that optimal algorithms do exist for scheduling independent tasks but these can only manage timing constraints and do not take into account any other priority information. Even in the simple case where tasks have only timing constraints, some optimal algorithms such as the earliest deadline prove to be very inefficient during system overload because it cannot recognise a failed process until its deadline has expired. This occurs because the earliest deadline algorithm has no knowledge of the run-times of tasks and hence it cannot predict whether a task will complete in time or not. Clearly then, a scheme involving the process laxities, where known, would be desirable. For sets of processes that have relative importance (real priority) as well as timing constraints, the problem becomes more difficult because the competing goals of priority and timing constraints are not always compatible. The simple optimal algorithms are unable to cope with this type of multiple constraint problem. Techniques do exist for transforming these systems into a single priority scheme (such as the priority transform for the rate-monotonic scheduler) but these are of limited use in a dynamic environment.

In a system with both priority and timing constraints, tasks of higher priority should always be allowed to run (providing they have a chance of completing before their

deadline) in preference to lower priority tasks. When the computation times of tasks are unknown, the agenda manager is limited to simply scheduling in priority order first and then by deadline within each level. When the timing information is known, the agenda manager has much more scope to construct an acceptable agenda and is able to avoid scheduling tasks that will be unable to complete successfully. This gives the system much better performance in overload situations because tasks that will fail can be rejected as soon as it is clear they cannot complete, preventing the task wasting CPU time. In a dynamic system, new processes are created at any time and the scheduling algorithm must be able to make a decision as to whether this task is schedulable or not. This function is performed by the *guaranteeing algorithm.* As the scope of the agenda manager is so limited when no computation time information is available, the following discussion considers the case where this information is available.

A guarantee algorithm must decide whether a new task can be added to the agenda manager in such a way that it will complete before its deadline and not jeopardise other tasks already guaranteed. A guarantee algorithm could be based on the earliest deadline schedule but this proves to be a poor solution due to the time required to assess schedulability. The earliest deadline algorithm is the simplest optimal algorithm but computationally it is expensive for testing schedulability. The tasks must be ordered in deadline order before stepping through time until either the process list is empty or a process violates its timing constraints. Especially in the general case where start-times are given, the cost of using the earliest deadline algorithm to verify each schedule (every time a process is added) is prohibitive.

As no suitable algorithms existed, a new algorithm has been designed to efficiently perform the guarantee. This algorithm uses a *slack time list* (STL) to store information

about the current schedule and enables a very fast guarantee. The STL is necessarily sub-optimal but it will be shown that the STL scheme performs very well under most conditions and will only demonstrate its sub-optimal characteristics under heavy load conditions. In such a state, the time overhead of the optimal earliest deadline scheme is very high and figures shown in Chapter 7 show that taking these timing details into account, the STL will actually perform better on average than the earliest deadline. The STL algorithm is particularly effective in multiprocessor situations where its speed of guarantee may allow tasks that would have failed with the earliest deadline algorithm to be scheduled elsewhere. The availability of extra processors also allows a large proportion of the falsely rejected processes to be successfully migrated. Another important aspect of the algorithm is that it is *fail safe* in that although it will occasionally reject tasks that it may have been able to schedule, it will never guarantee a task and fail to provide it with adequate resources to complete successfully.

5.5 The Slack Time List Algorithm

The STL algorithm works by deciding quickly whether or not a new process can be successfully scheduled. If it can be, the task is added to the agenda and scheduled according to any optimal algorithm such as the earliest deadline.

The STL algorithm maintains a list of 'slack time' contained in the current schedule. The method will become clear during the explanation of the algorithm but it is useful to imagine this slack time list as a list of structure elements as shown in Figure 5.3. The list is ordered by decreasing `time` and each element specifies the total slack time before `time` (`slack_total`) and the slack time between this element and the next

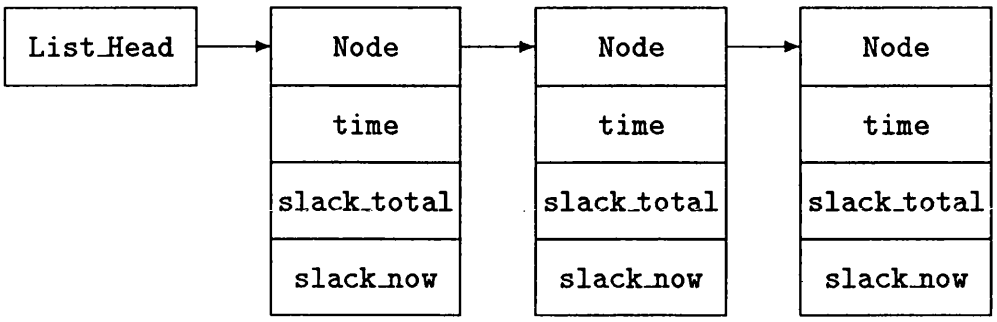

Figure 5.3: The Slack Time List (STL)

(`slack_now`). In the following text, the STL elements are referred to as a 3-tuple (`time,slack_total,slack_now`). A worked example is included after the basic explanation. The algorithm proceeds as follows.

1. Initialise the STL to contain a single element [(0,0,0)]. Initialise a list to contain the agenda.

2. Receive the list of new tasks from the dispatcher, ordered first by priority and then by minimum laxity.

3. Take the first task from the task list and add it to the schedule list provided that $d_0 < s_0 + c_0$. Add the element (d_0, l_0, l_0) to the head of the STL, where d_0 is the deadline of the task and l_0 is the laxity. The STL then shows that there is l_0 time free between d_0 and 0.

4. This step is repeated while tasks remain on the task list and when the algorithm is called subsequently. If the next task, T_i, can schedule, there must be c_i seconds (computation time) available between the task's start-time, s_i, and the task's deadline, d_i.

(a) Calculate the time, r, that must be recovered from the STL and find the earliest node in the STL, S_1, which has a time value greater than the deadline of the task.

If $d_i < S_{\text{head,time}}$

$$r = c_i \text{ and } S_1 : S_{1,\text{time}} > d_i$$

otherwise,

$$r = c_i - (d_i - S_{\text{head,time}}) \text{ and } S_1 : S_{1,\text{head}}$$

(b) If $r > S_{1,\text{slack\_total}}$ the task cannot be scheduled and is **rejected**. If $r <= 0$ then the task can be **guaranteed**. Also, if $r < S_{1,\text{slack\_total}}$ and the start-time of the task is immediate then the task can be **guaranteed**.

(c) As the start-time is not immediate, find the *last* element in the STL for which `time` $> s_i$. Call this element S_2

(d) Find the total time available between the task's start-time and deadline. This is given by,

$$t_{av} = S_{1,\text{slack\_total}} - S_{2,\text{slack\_total}} + \max(S_{2,\text{time}} - s_i, S_{2,\text{slack\_now}})$$

(e) If $t_{av} < r$ then the task is **rejected** because the slack that does exist in the schedule occurs too early to be used by this process. Otherwise, the task *will* schedule and is **guaranteed**.

(f) If the task has been guaranteed, it must be added to the agenda and the STL must be updated to remove the time that is now allocated. During this update, new elements must be added if the deadline or the start-time of the new task are unique times for which no STL element exists. Any remaining elements earlier than the present time can be removed as well as

any redundant elements caused by the removal of slack time. For example, a list [(20,7,3), (15,4,4), (8,0,0)] will become [(20,0,0), (17,0,0), (8,0,0)] after scheduling a task $d_i = 20, c_i = 7, s_i = 4$ and this can be reduced simply to [(20,0,0)].

The resulting schedule list then contains a set of tasks that *can* be scheduled. Any optimal algorithm can then be used to actually schedule the guaranteed task set. For efficiency, guaranteed tasks are added directly on to the agenda in deadline order and a simple earliest deadline policy is used by the dispatcher.

Worked Example

Consider the problem of generating a schedule for the following 5 tasks that are already ordered in terms of priority and laxity.

Task	Priority	Start Time	Comp. Time	Deadline	Laxity
T_0	10	8	10	20	2
T_1	9	30	4	35	1
T_2	8	0	10	25	15
T_3	8	0	20	40	20
T_4	8	5	14	40	21
T_5	7	3	2	10	5

The scheduling proceeds as follows.

1. The STL is initialised to [(0,0,0)].

2. As T_0 has a positive laxity and the STL is empty, it can be scheduled. This creates a schedule list containing [T_0] and the STL becomes [(20,10,2), (8,8,8), (0,0,0)].

3. T_1 is considered. As $d_1 > 20$ (the highest `time` value on the STL), r is calculated as $c_1 - (d_1 - 20)$. As this is negative, the task can be scheduled. The schedule list becomes $[T_0, T_1]$ and the STL becomes [(35,21,1), (30,20,10), (20,10,2), (8,8,8), (0,0,0)].

4. T_2 must find its entire c_2 from the STL as d_2 is less than 35. Find the elements between which the time must be allocated, which in this case is $(30, 20, 10)$ and $(0, 0, 0)$. Clearly there is enough time here to schedule the task so the schedule list becomes $[T_0, T_2, T_1]$ (Note the deadline order). After the addition of this task, the STL will be [(35,11,1), (30,10,5), (25,5,0), (20,5,0), (8,8,8), (0,0,0)]. Whenever two consecutive elements have zero `slack_now`, the earlier one can be deleted as it contains no useful information so the list reduces to [(35,11,1), (30,10,5), (25,5,0), (8,8,8), (0,0,0)].

5. Attempting to add T_3 reveals that $c_3 - (d_3 - 35) = 15$ time units must be found from the STL. On inspection however, only 11 units are available so this task cannot be scheduled.

6. T_4 requires $c_4 - (d_4 - 35) = 9$ time units from the STL which are available. This task must recover 9 units between the 35 and 9 elements but this is possible so the task is scheduled. The schedule list then becomes $[T_0, T_2, T_1, T_4]$ and the STL becomes [(40,2,0), (35,2,0), (30,2,0), (25,2,0), (8,2,0), (5,2,2), (0,0,0)] which reduces to [(40,2,0), (5,2,2), (0,0,0)].

7. The final task requires two time units before the task deadline of 10. As the start-time is 3, the task fits into the remaining time in the schedule. The final schedule is now $[T_5, T_0, T_2, T_1, T_4]$ and the STL becomes [(40,0,0), (5,0,0), (0,0,0)]. The latest element with a zero `total_slack` time is the earliest point at which any

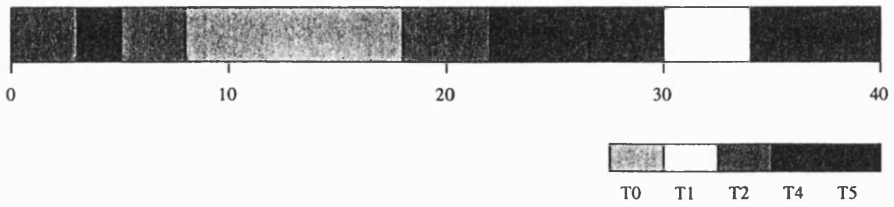

Figure 5.4: The Deadline Scheduling of the Worked Example

new task can be scheduled. The list therefore reduces to [(40,0,0)].

The algorithm has now finished and the schedule list, $[T_5, T_0, T_2, T_1, T_4]$, can be implemented by any optimal algorithm. The tasks here have been added to the schedule in order of increasing deadline. A simple deadline scheduler can now implement this schedule as shown in Figure 5.4. As the schedule becomes full, the STL size reduces, avoiding the undesirable property of the earliest deadline guaranteeing algorithm where its work load increases with the number of processes and its performance degrades at exactly the time when it needs to be at its most efficient.

The STL algorithm is sub-optimal because it will occasionally falsely reject a task that an optimal algorithm could have scheduled. The reason for this behaviour is that the STL algorithm must commit the slack time held in the list between pairs of STL elements under conditions where new tasks overlap the end of one task and the beginning of another. Once the slack is committed, the scheduler can be 'broken' if a new task requires that the slack time had been placed elsewhere. In practice, this sub-optimal behaviour only becomes apparent under heavy load conditions where the speed of the algorithm continues to outweigh the effect. The algorithm as described tends to allocate the slack-time earlier rather than later in the list when an arbitrary choice has to be made. This choice means that urgent processes will normally be scheduled optimally and the small number of false rejections will tend to be for processes with

future start-times that have time to be successfully migrated to another processor.

The STL algorithm described so far relies on knowing the computation time of each task. For systems where this is not known, it is very hard to provide any sort of sensible scheduling algorithm other than the earliest deadline system. Unfortunately, in multiprocessor systems it becomes impossible to know in advance whether or not a task will schedule and hence decide whether the task should be migrated. Under these conditions, the best solution possible is some form of load-balancing system that attempts to maintain a roughly equal distribution of tasks around the system. The severe limitations of such an approach makes in unsuitable for critical real-time systems. In practice, some idea of the computation time of individual tasks is usually obtainable and can help drastically to improve the system performance.

In systems where the computation times of certain tasks are not known, most very critical tasks will have known computation times and also have a high priority. In these circumstances, the STL algorithm can be used to schedule these tasks until the agenda manager reaches a task in the process list that does not have a defined computation time. Once this process has been scheduled, it is impossible to know the system state beyond that point but a slightly enhanced earliest deadline system can be used. After the STL has been used, the earliest possible start-time of the next task is clearly the `time - slack_total` of the head element of the STL. Maintaining a notion of this absolute best case start-time, updating it whenever tasks of known computation time are scheduled allows a slightly improved schedule because some tasks that could not start before there deadline can be rejected. This scheme is still very uncertain so wherever possible, the computation times of tasks should be used.

5.6 Multiprocessor Scheduling

The multiprocessor scheduling scheme, as already suggested, relies on the ability to guarantee tasks locally using the STL. Any task that cannot be guaranteed is then handled by the global scheduler that forms part of the dispatcher on each processor. A process that must be migrated is sent to some other processor according to the protocol in use. A number of protocols were tested, including a no-migration (control) scheme and two random schemes. The two 'real' algorithms used were a bidding scheme where the task details are sent around the system and processors return bids reflecting their ability to schedule the process, and a scheme where the process is sent directly to another processor based on load information that each processor broadcasts around the system, sometimes called focussed addressing. These schemes are described in detail in Chapter 6 and their performance is shown in Chapter 7.

For the bidding schemes, the STL algorithm is able to give a useful measure of a processor's ability to schedule a new task. When the task information arrives, the STL procedure is invoked as if the task was to be added. If the task cannot be scheduled, there is no need to return a bid. If the task does schedule, a measure of the processor's spare capacity in scheduling this particular task can be approximated with the ratio of t_{av} (from Page 107) to the task's computation time. The ratio effectively indicates how many tasks similar to this one could be scheduled by this processor under the current load conditions. This scheme proves to be both effective and efficient as the bidding overhead is kept low.

The load information used in the focussed addressing scheme is far less precise than the bidding information. The simple scheme chosen broadcasts just two numbers, the

`time` and the `slack_total` of the STL's head element. The broadcasts are sent at a specified interval and are used as a rough guide to the state of processor usage. This simple scheme works well in practice despite relying on approximate and somewhat out of date information.

5.7 Consistency of Shared Data

Data consistency is a major problem with shared data. Consider a rule that performs some function f given that x is true. If x is true and this rule fires but in the mean time x has become false, then f will be performed inconsistently. The basic problem occurs because the data is not guaranteed consistent during the execution of any single rule.

On a single processor system using co-operative scheduling, data consistency is not a problem. As scheduling is not preemptive, and the CPU is given up only between rule firings, the data is guaranteed consistent. However, in the multiprocessor case the problem arises again. In this system, the problem is tackled with locks. When critical data is held in a shared fact base, the fact base must be exclusively locked before running the rule. The choice of locking granularity is important to the performance of the system. If each data item is locked individually there will be high traffic rates of tasks attempting to obtain (and release) locks whereas if there are too few locks, inference engines will be forced to lock more data than they really require and cause unnecessary blocking. The solution chosen for this system is to lock individual fact bases so the balance described above can be adjusted by the developer.

It would be desirable to have a Truth Maintenance System running across the network to formally validate the distributed data. For this project, the implementation of such

a system would be impractical so care must be taken to ensure that the rules and fact bases are written in such a way as to ensure data consistency problems do not arise.

5.7.1 The Locking Mechanism

Shared data can become corrupted if multiple processes have unchecked access permission. This problem can be overcome using semaphore locks. Every fact base in the system has the facility to be locked as a whole. The idea of locking individual data elements soon causes an abundance of lock requests and can easily lead to deadlock problems if a number of sets of data need to be locked before rules can proceed.

The fact base locks are not used automatically by the inference engines as it is inefficient to lock data used solely by one inference engine or if it can be guaranteed that for some other reason exclusion will be enforced. In the development system, locks are written explicitly into rules which necessitates careful planning of fact bases and an appreciation of the inter-dependencies of co-operating inference engines. It is possible, even with quite large knowledge bases, to organise the fact and rule bases to minimise these locking problems. However, for complex systems it would be desirable to develop a more integrated compiler that could automate this process.

5.8 Priority Inversion

Any locking scheme within a priority based scheduling scheme can cause an effect known as *priority inversion* which occurs as follows. Consider a system where a low priority process (L) runs and obtains a lock before being preempted by a higher priority process (H). If H then attempts to obtain the same lock, it will fail and be forced to

wait for the lower priority process to release the lock. This is not priority inversion but merely the effect of mutual exclusion but consider now a third process with a priority between the other two (*M*). If *M* becomes active while *H* is waiting for *L* it will preempt *L* and therefore further delay *H*, the high priority process. This effect is known as *priority inversion* and can in the worst case cause unbounded delays on the high priority process.

Priority inversion can be solved quite simply using a number of schemes. One solution is for the process that has obtained the lock to temporarily inherit the priority of any other higher priority process that attempts to obtain the same lock. This scheme known as priority inheritance would have meant that *L* in the above example would have assumed the priority of *H* for the duration of the lock and hence avoided being descheduled by *M*. A similar scheme known as the ceiling protocol immediately raises the priority of a process that obtains a lock to that of the highest process that ever accesses that lock. The ceiling protocol is simpler than the inheritance protocol to implement but does lead to higher priority tasks being blocked even when they do not access the lock. An even simpler scheme is to make the code contained between two locks non-preemptive. Provided that the critical sections remain small this scheme provides an acceptable solution and fits well with the co-operative scheduling scheme employed in this project. As such, accesses to locks must be contained within a single scheduling step. To avoid the potential overhead of waiting for remote locks to become free, a method described in the following chapter allows the dispatcher to allocate the CPU to another task while the lock is being obtained.

5.9 Summary

This chapter has outlined the main design issues of Grape. The architecture of the system has been proposed and each of the components has been discussed. The following chapter will go on to describe the implementation of this system on a shared memory transputer system and this will help to clarify some of the ideas presented in this chapter.

Chapter 6
Implementation of the Grape System

6.1 Implementation Language

Traditionally most AI systems, including expert systems, have been developed with symbolic languages such as Lisp or Prolog. Although these languages were considered for Grape they were rejected in favour of a mixture of ANSI standard C and C++.

Lisp is a functional language (with side-effects) and has been used extensively for developing expert systems. OPS5 for example is implemented in Lisp. The language was designed for list processing and this representation remains at the core of the language. Prolog is a declarative logic language that solves problems expressed in first order predicate logic. A description of these languages and their application to expert systems is beyond the scope of this thesis but the justification for rejecting these languages is similar.

- For portability, it is essential that there is a single standard implementation of the development language and that a compiler for the language exists on a wide variety of machines. Both Prolog and Lisp exist in many dialects and are implemented on only a limited range of machines. Neither Lisp nor Prolog is supported by Helios.

- For a real-time environment where speed is important, an efficient and predictable

language is essential. Prolog and Lisp are usually much slower than imperative languages for a number of reasons. Both languages were traditionally interpreted and although most recent implementations are compiled, many existing platforms are still interpreted. Even compiled versions lack the speed of languages such as C, partly because of the immaturity of the compiler technology and partly because of the languages themselves. Both Lisp and Prolog are typeless which introduces an overhead in dynamic type-checking and symbolic manipulation and both languages rely heavily on dynamic memory allocation. Rather than keep track of every item of memory, the run-time systems continue until memory becomes low and then perform *garbage collection.* This type of behaviour is disastrous for real-time systems as there is no way of knowing when this will happen. The use of large amounts of memory (typically many mega-bytes) also realistically limits these languages to machines with virtual memory—a facility not available under Helios.

- The system is to be implemented under Helios and must, for portability, run under Unix. Both of these operating systems are written in C and provide extensive support for it, including a compiler, development tools and the standard libraries. Although C++ is usually not provided as standard, it is available under Helios, most versions of Unix and is increasingly common on all platforms. The GNU project provides an excellent C and C++ compiler that is available free and is extremely portable.

- The diesel engine simulation is written in C so interfacing to it is made simpler by the use of a common language.

- Some aspects of the Bath transputer system require low level access to the system

because they are not directly supported by the operating system. C allows direct memory accesses to specified locations allowing the hardware to be used as required—this is not possible under Prolog or Lisp.

- The difficulty in developing knowledge-based systems with a standard procedural language such as C is overcome to a large extent by the use of C++. C++ provides an object-oriented environment that allows knowledge to be represented in a more natural manner and much of the detailed implementation to be hidden from the higher level procedures.

The inference engine, including the internal knowledge representation is written entirely in C++. The compilers make use of the `flex` and `bison` utilities[1], with the remaining code written in C. `flex` produces the C source for a lexical analyser from a specification of the language tokens, while `bison` takes a context-free grammar as input and produces the source code for an appropriate parser. These tools simplify the process of converting the source language (KRL) into the code tree on which the semantic verification is performed before the target language is generated. The scheduler, including both the dispatcher and the agenda manager are written in C.

6.2 Portability

Grape has been designed (and implemented) to be portable. The entire system is written in ANSI standard C and C++ as defined by Stroustrup [98] which forms the basis of the proposed ANSI standard. The software relies as little as possible on the underlying hardware and operating system. The majority of the system specific routines are used

[1] These are the GNU project equivalents of `lex` and `yacc` respectively.

by the dispatcher to create, control and kill processes and to set up the shared memory areas for use by the inference engines. The process scheduling is controlled using semaphores, supported by most multitasking environments, removing the need to rely on the underlying scheduling mechanism of the operating system.

The Grape system currently runs under Helios, BSD 4.3 Unix, SunOS 4.x, AT&T's System V Release 4 (SunOS 5.x and IRIX 4.0.5), Linux 0.99 and MSDOS. The inference engine can be optionally compiled as a stand-alone executable and this should be possible on any system with either an ANSI C compiler or a K&R compiler (the MSDOS version is compiled in this way as it cannot support multitasking). Currently only the Helios version supports multiprocessors.

6.3 The Dispatcher

6.3.1 Building a portable scheduler

The dispatcher relies, more than any other module, on the underlying operating system. For the dispatcher to carry out its job it must be able to control which process is allowed to run at any given time. Unfortunately, standard Unix with its fair-share scheduler and the Helios Operating System relying on the transputer's hardware round-robin scheduler do not allow this level of process control.

To overcome this fundamental problem, a co-operative scheduling scheme has been developed. The scheduling mechanism relies on a set of semaphores (see Section 6.6) that are used to dictate which process will be allowed to run at any given time. The processes themselves have to be aware of the scheduling protocol but the code modi-

fications are small and have been encapsulated in a small number of simple functions. The advantage of the scheme is of course that the dispatcher is able to control the scheduling of the system and therefore determine the temporal response of the system. A co-operative scheme, relying only on semaphores will run not only under both Helios and Unix but also under any other operating system that provides this basic support.

The dispatcher maintains a set of Process Descriptors (PD's) of the form shown in Figure 6.1. There is one PD for each process under its control and it is used to store the process details. Some of the process descriptor is shared with the process concerned.

Each PD contains a unique run permission semaphore that the dispatcher uses to give control to the chosen process. The slave processes block by waiting on this semaphore and are able to run only when the semaphore is signalled by the dispatcher. The `Control` element of the PD contains a structure shared by all local processes (see Figure 6.2) and includes another semaphore, `done`. After scheduling a selected task, the dispatcher blocks itself by waiting on `done` and the selected process may run for a short time period, wake up the dispatcher by signalling `done` and wait again on its run permission semaphore. If the running period of each process is kept small and special care is taken to ensure the CPU is not left idle during disk I/O etc., the performance of such a scheme is comparable to a true preemptive scheduler, except in the worst case response time. Clearly, if a vital action needs to be performed, a preemptive scheduler could stop the running process immediately and schedule the new process whereas the co-operative scheme must wait for the running process to give back the CPU. If the CPU has only just be given to the process, the time from the event occurring to the new process being scheduled could be considerably longer than in the preemptive case but if the time-slices are kept small this should in most cases be acceptable.

```
typedef struct ProcDesc
{
    dll_Node          node;          /* To attach to list */

    int               procArgc;      /* Process Identification */
    char**            procArgv;
    int               procNumber;
    int               parent;        /* procNumber of parent */

    Semaphore         runPerm;       /* Process Control */
    Status            status;
    Control*          control;
    int               finished;
    struct ProcDesc*  waiting;       /* to hold the ProcDesc* of */
                                     /* waiting process */
    int               result;        /* passed back to the parent */
    int               childResult;   /* ... in this variable */

    ProcType          type;          /* process type and mode */
    ProcMode          mode;
    int               startTime;     /* timing information */
    int               compTime;
    int               priority;
    int               deadline;
    int               remaining;
} ProcDesc;
```

Figure 6.1: The Process Descriptor (PD). The actual PD structure contains a number of other fields for process accounting etc. that are omitted here for clarity.

The `ProcType` field stores the timing class of the process, and can take the values of `HARD_REAL_TIME` for tasks that are worthless beyond their deadlines, `SOFT_REAL_TIME` for tasks that have a reduced utility value after their deadline, `PERIODIC` for tasks that must be performed once every T seconds or `ASYNCHRONOUS` for non-real-time tasks. The `Status` field indicates whether the task is `NOT_CREATED` because it has never yet been scheduled, `RUNNING`, `SLEEPING` or in the special state `LEAVE_ME` (explained later). When a child process is spawned, its parent process can either continue to operate (concurrently with the child), or it can suspend and wait for the child process

```
typedef struct Control
{
    Semaphore done;
    ProcMode  childMode;
    ProcType  childType;
    int       childPriority;
    int       childArgc;
    char**    childArgv;
    int       childStart;
    int       childComp;
    int       childDead;
    Pool      pool;              /* facility to allow */
                                 /* shared memory */
    void*     mem;
    int       newAgendaRequest; /* set to the priority */
                                 /* value of the request */
} Control;
```

Figure 6.2: The Control Structure

to complete. When a new process is spawned, the `ProcMode` field will be set to either `WAKE_PARENT` or `DONT_WAKE_PARENT` depending on the parent's wishes. If the parent does suspend, its process descriptor is removed from the process queue and held in the child's `waiting` field.

The `control` structure, shown in Figure 6.2, contains the `done` semaphore, fields to allow new child processes to be created, the `newAgendaRequest` field and the shared memory pool (the need for which is discussed later). If a process wishes to create a new child, it uses the `control` structure to inform the dispatcher of the process details. It then sets its own `ProcMode` depending on whether it wants to run concurrently with the child or to wait for it to complete and returns control to the dispatcher. The spawning is performed by the dispatcher so that the task can be properly incorporated into the co-operative scheduling scheme.

The `newAgendaRequest` field is use to inform the dispatcher that a process priority has been changed during this task's execution. The ability to change process priorities allows high level control of the system behaviour but changes need to be alerted to the dispatcher so it can, if necessary, invoke the agenda manager. The `agendaLevel` variable discussed in Chapter 5 is maintained by the dispatcher to reflect the highest priority event so far that requires a new agenda. When a process changes either its own priority or that of another process, the `newAgendaRequest` is set to the higher value of the process's old and new priorities. The dispatcher then sets its `agendaLevel` to be the higher of its old value and the new value of `newAgendaRequest`.

When a new task is created, the dispatcher must incorporate the task into the scheduling scheme and pass it a pointer to its new process descriptor. Tasks created under Helios inherit a port descriptor (analogous to an `occam` channel) that can be used to pass messages back to its parent task. The child must then create a new port (on which it can receive messages) and pass the address of this back to its parent using the inherited port. This new port is then used by the parent to send the child the address of its PD. Once this initial message passing phase is completed, all task communications are performed using shared memory. This procedure is encapsulated in a library function to simplify the modification of co-operating process. (see Figure 6.3).

6.3.2 Other Dispatcher Functions

The dispatcher performs a number of other functions mainly concerned with task management, such as keeping track of computation time remaining and storing the accounting information of task completions and failures. In the single processor version of the Grape system, the dispatcher is the root process and hence is required to

maintain the shared fact base table and similar functions that are normally performed by the system loader. A command-line interface (shell) is also provided by the dispatcher that can be used to interrogate the scheduler at run-time. Information such as the current agenda and information of past and present tasks can be obtained from this interface. In the multiprocessor version, this command-line interface is again provided by the system loader.

6.3.3 Optimisation of I/O Operations

A significant penalty of a co-operative scheduling scheme is the removal of latency hiding. In most multitasking computers, when a process performs a disk access or other similar slow but non-CPU intensive task, the operating system is able to schedule another process while the first is waiting for the operation to complete. This ability to utilise the CPU in otherwise idle periods greatly increased processor usage and system performance. In a simple co-operative scheme, this advantage is immediately lost.

To overcome this problem, a new state was introduced called `LEAVE_ME`. When disk I/O or other similar operation is performed, the process sets its `Status` field (in the PD) to `LEAVE_ME` and signals `done`. It then proceeds with the operation before resetting the status to `RUNNING` and waiting on its run permission semaphore. The dispatcher looks at the process status and will never schedule a process in the `LEAVE_ME` state. Using this scheme, latency hiding is once again possible and the dispatcher remains in control of the CPU. The CPU consumption between the signalling `done` and waiting on `runPerm` is very low and does not interfere with the other running process. This enhancement of the co-operative scheduler produces dramatic performance gains for even relatively low levels disk I/O.

```
#define waitForStart(pd) Wait(&(pd)->runPerm)
#define giveup(pd)       Signal(&(pd)->control->done); \
                         Wait(&(pd)->runPerm)
#define terminate(pd,x)  pd->result = (x); \
                         pd->finished = 1; \
                         Signal(&(pd)->control->done);

extern ProcDesc *getProcDesc(void);

#define doIO(pd)         (pd)->status = LEAVE_ME; \
                         Signal(&((pd)->control->done))
#define doneIO(pd)       (pd)->status = RUNNING; \
                         Wait(&(pd)->runPerm)
```

Figure 6.3: Co-operative Scheduling and Latency Hiding Primitives

6.3.4 Adding Tasks into the Scheduling Scheme

Inevitably, the use of a co-operative scheduler necessitates some minor modifications to the processes under the scheduler's control. All the necessary procedures are contained in a small set of macros and library functions shown in Figure 6.3. The steps involved in modifying a process are summarised below.

- Add `#include "proclib.h"` in the code preamble of the module containing `main()` and other modules that require the scheduling functions. This header defines the above macros along with the `ProcDesc` and `Control` structures.
- Define a global variable `ProcDesc *pd`.
- Insert the line `pd = getProcDesc();` at the top of `main()`. This function performs the initial message passing and receives the address of this process's PD from the dispatcher.

- Insert the line `waitForStart(pd);` before the program proper starts. This simply waits for the run permission semaphore to be signalled.
- At regular intervals, add `giveup(pd);` a macro containing the instructions to return control back to the dispatcher. The scheduling overhead is low so these should be inserted quite frequently to ensure a responsive system. In most programs, a central loop exists meaning that only a small number of these instructions actually need to be inserted.
- When disk I/O is performed or similar operations such as user input, enclose the operation between `doIO(pd)` and `doneIO(pd)` macros to allow the dispatcher to schedule another task while the I/O is performed.
- As the process is about to end, it should call `terminate(pd,x)`, where `x` is the return value of the task that will be passed back to its parent.
- Finally, the code should be compiled and linked with the module `proclib.o` which contains the definition of the `getProcDesc` function as well as the functions needed for processes to spawn child tasks.

6.4 The Agenda Manager

The agenda manager is called with a list of new tasks that are, if possible, to be added to the local agenda. The STL algorithm is used to quickly determine if each task is schedulable on this processor. The majority of tasks will normally schedule locally but some will be rejected. If the task is rejected because it can no longer be scheduled (with a negative laxity) then the agenda manager can do nothing further. If however the task is rejected because the local node does not have the required resources (time

for example) the agenda manager will attempt to migrate the task to another processor where it can be scheduled.

The mechanism for deciding which remote processor should receive the task depends on the protocol in use. Several protocols have been compared in this work: a random policy; a bidding scheme and a focussed addressing scheme. With the random policy, the task is sent to a random node in the system. In the bidding scheme the agenda manager sends out the details of the new task to each processor and they return bids depending on their ability to schedule such a task. The task is eventually migrated to the highest bidder. The focussed addressing scheme relies on each agenda manager regularly broadcasting its present load status. When an agenda manager wishes to migrate a task, it chooses the processor with the greatest capacity based on these load figures.

The bidding scheme is accurate but carries a relatively high overhead whereas the focussed addressing scheme is less accurate but much faster. More details about all these algorithms and a comparison of them under various system loads is shown in Chapter 7.

All communication between agenda managers is by passing message packets across the backplane. Each agenda manager has an incoming work queue and the location of each node's work queue is known to every agenda manager. Packets are sent by obtaining a buffer from the remote node's free buffer pool, transferring the message to it and then attaching this buffer to the remote queue. Obtaining the free buffer and attaching it on to the queue must be performed atomically so there are two locks to ensure mutual exclusion. As the two locks are distinct and there is no need to hold a lock during the message transfer stage, the contention for the queues is minimised.

6.5 The Inference Engine

6.5.1 Knowledge Representation

The fact bases are stored internally as shown in Figure 6.4. Each base contains the definition of each object defined in it, including the object name, the definition of each fact that makes up the object, and includes a set of instances of that object. The individual facts may contain a single value or (optionally) a number of values. Each value consists of the data and a confidence level along with a time-stamp and a recency number. The addition of the time-stamp field allows the explicit use of time in the inferencing process. In the default case only one value will exist for each fact but it is possible for a number of values to be held simultaneously. Additionally, facts may be declared for trend analysis which allows a specified number of old values to be retained allowing the dynamics of a value over time to be considered during the reasoning process.

6.5.2 Sharing Fact Bases

The transputer does not support memory protection so sharing data is straight forward. A process may simply allocate an area of RAM and pass its address to any other process that requires access to it. Problems do arise however when data must remain in memory after the allocating process has terminated. Under Helios, each process maintains a memory *Pool* that is used to keep track of all the memory it allocates. When a process terminates, the Helios kernel uses the process's Pool to deallocate the memory reserved by that process. The Grape fact bases are allocated by the first process to access them but they must remain resident until all accessing processes terminate,

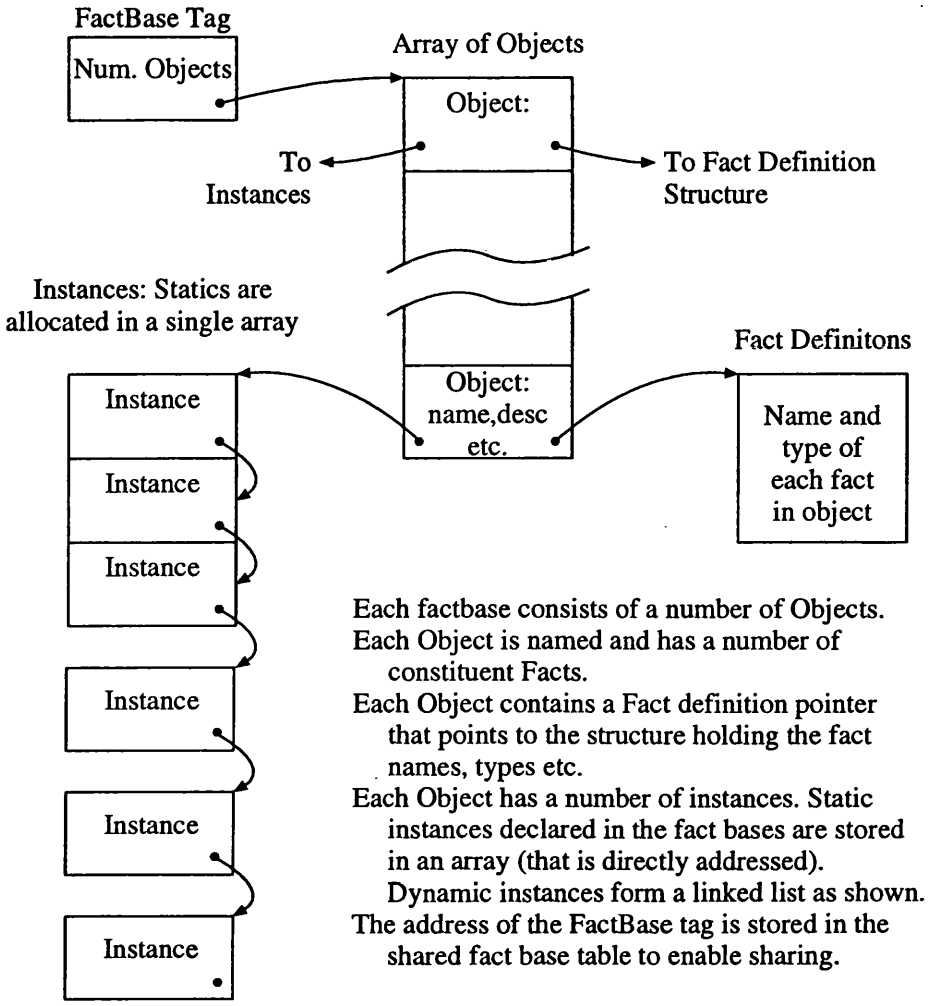

Figure 6.4: The Internal Fact Base Structure

even if the allocating process has finished. This functionality has been achieved by maintaining a dedicated pool on each processor that is initialised by the dispatcher on start-up and is passed, in the process descriptor, to each process. Whenever processes need to allocate shared memory rather than private memory this global pool is used via a set of primitives, `SysNew()`, `SysDelete()` and `SysStrdup()`.

Allocating memory from a central pool rather than the process pool complicates the inference engine because C++ polymorphism cannot be used. If it were not for the

central pool, facts of each type (int, float, string, etc.) could be created and stored in an array of `fact*` pointers. Using C++ polymorphism, the facts could then be treated identically, regardless of their true type because C++ will resolve the bindings automatically at run-time. Unfortunately, the information used to perform this run-time binding is provided by the `new` operator that allocates memory from the process's private pool. For the shared fact bases therefore, explicit type information has to be included into the fact class definitions.

6.5.3 The Fact Base Table

Each fact base may be either private or shared. Each rule base contains a list of fact bases which it requires and the individual inference engines must ensure that these are available. A central hashed lookup table is maintained to store the current status of all the fact bases in the system. Initially, the fact base table is empty and as inference engines load in new fact bases, they are entered into the table. The table is guarded by a semaphore to ensure that mutual exclusion is enforced.

If a rule base requires some particular fact base, it first looks it up in the fact base table. If the base is marked as loaded, the user count for that fact base is incremented and the fact base becomes shared. If the fact base is not yet loaded, it is marked as `LOADING` while the load takes place. Once loaded, the fact base table status is changed to `LOADED` and the root address of the fact base is entered into the table.

When inference engines terminate, they decouple themselves from their fact bases by decrementing the user count in the fact base entry. If the user count becomes zero, the fact base may be unloaded depending on its cache status. On system startup, fact

bases may be declared cached so they are never unloaded. Similarly, fact bases can be preloaded to avoid the overhead of loading in important fact bases.

6.6 Locking on the Bath Transputer System

Mutual exclusion is required for certain fact base accesses and for the manipulation of the shared global scheduling queues. This is provided using *semaphores*. A semaphore is a non-negative integer counter that is initialised to a number equal to the number of free resources available (1 for a simple lock). The semaphore is accessed with two special functions, `Wait()` and `Signal()`.[2]

`Wait()` atomically decrements the counter (to claim one of the free resources) and `Signal()` atomically increments the counter when the resource is subsequently freed. When the counter becomes zero, showing that there are no free resources left, tasks that `Wait()` on the semaphore will block until the counter becomes greater than zero. The order in which blocked processes are woken following a `Signal()` is undefined but is usually on a FIFO basis. The most important property of the semaphore is that these operations are atomic so if the counter is set to one and two processes `Wait()` on the semaphore, only one will succeed immediately, the other will be blocked until the semaphore is signalled again. A semaphore can be used as a resource lock simply by initialising it to one and enclosing each access to the resource within a `Wait()` and `Signal()` pair. Each resource that needs to ensure mutual exclusion is therefore guarded by a semaphore that is initialised to one.

[2]Dijkstra [43] originally used the functions *P()* and *V()* to represent Passeren (pass) and Vrijgeven (release) in his native Dutch.

For efficiency, the mutually exclusive locking protocol must provide a process blocking facility rather than relying on spin-locks. In most multitasking operating systems the semaphore implementation is blocking in that a process which fails to obtain the semaphore is descheduled and queued on an event queue. A blocking scheme ensures that processes waiting for locks do not consume any CPU time unlike an inefficient busy-wait protocol.

For single processor systems the Helios system semaphores are adequate to provide the mutually exclusive locking functions. In multiprocessor systems the problem immediately becomes more complex. The Helios semaphores are guaranteed atomic because the locking code is run at high priority on the transputer and hence cannot be interrupted. Also, when semaphores are already locked the local kernel deschedules any process attempting to obtain a lock and queues it on the semaphore queue. This assumes that the waiting process is under the control of the local kernel and therefore is a local process.

The fact that semaphore operations must be local necessitates some other mechanism to implement remote resource locks. The way this is achieved is by using the event mechanism on the Bath transputer system and providing special backplane signal and wait routines, `BPSignal()` and `BPWait()`. As mentioned in Section 4.1.3, each processor node of the Bath transputer system contains an event register that causes a hardware event signal when data is written to it. An event handler must be installed on each processor to handle these events that clears the event by reading the event register.

`BPSignal()` can be implemented by writing the address of the semaphore into the remote processor's event register and writing the event handler to simply read this value (thereby clearing the event) and performing a local `Signal` on that address.

```
extern void BPSignal( Semaphore *sem )
{
    int SemBase = GetProcBase(sem);

    if((SemBase == LocalNum) || (SemBase==GetGlobalNum)) {
        Signal(sem);
    } else {
        ProcRegs *pr = GetProcRegs(SemBase);
        pr->EventReg = (word) sem;      /* asserts the remote */
    }                                   /* event pin */
}
```

Figure 6.5: Implementation of `BPSignal()`

As the event register is guaranteed atomic in hardware, the backplane signal is also guaranteed to be atomic. The code for the `BPSignal()` is shown in Figure 6.5.

Implementing `BPRemoteWait()` is more complex because the Helios `Wait()` *must* be issued locally for the atomicity and the blocking to perform correctly. To overcome this problem the protocol shown in Figure 6.6 was developed.

The issuing processor creates a temporary semaphore and waits on this after passing its address to the remote processor. The remote processor in turn forks a process to wait on its local semaphore and when the signal arrives it performs a `BPSignal()` back to the original processor. The remote wait code is shown in Figure 6.7. For the remote wait, the local and the remote semaphore addresses must be given to the remote processor. These are stored in two registers in the special area at the top of the local memory map. The event is caused by writing a zero into the remote event register which is identified by the event handler as a `BPRemoteWait()`. As these registers may be written to by any processor in the system they must be locked using the processor's test-and-set flag (see Section 4.1.3). The TAS flag is obtained in the local processor using a busy wait

Local Processor		**Remote Processor**
• Create a temporary semaphore *s*.		
• Wait for the TAS of the remote processor.		
• Send an event with the address of the remote semaphore *r*, and the temporary semaphore *s*.	⟹	• Catch and clear event and recognise it is a `BPWait()`.
• `Wait(s)`		• Read the semaphore addresses (*s* and *r*) and free the TAS • `fork()` a process to deal with the wait. *(—in the forked process—)* • `Wait(r)` (a local operation)
• Continue.	⟸	• `BPSignal(s)` • exit.

Figure 6.6: A Remote Blocking Semaphore Protocol

and is cleared by the remote event handler after the two semaphore addresses have been read. The busy wait on the remote TAS is unavoidable but the locking period is short and this does not cause an excessive overhead.

On each processor, an event handler must be installed to deal with the events as they occur and to clear the event register. To implement the protocol shown earlier, the event handler in Figure 6.8 was developed. An event will either be a remote `BPSignal()` or a `BPRemoteWait()`. This is determined from the event value and the appropriate action is taken. If the event was caused by a `BPRemoteWait()` then the TAS flag has to be cleared after reading the semaphore addresses.

```
extern void BPRemoteWait( Semaphore *sem )
{
    int SemBase = GetProcBase(sem);

    if((SemBase == LocalNum) || (SemBase==GetGlobalNum)) {
        Wait(sem);
    } else {
        ProcRegs *pr = GetProcRegs(SemBase);
        volatile word *flag = &pr->TASFlag;

        Semaphore temp;              /* local semaphore */
        BPInitSemaphore(&temp,0);

        while(*flag<0)               /* Busy wait to lock */
            continue;                /* the processor */
        pr->localSem = (void*) sem;
        pr->remoteSem = (void*) MakeGlobalAddr(&temp);
        pr->EventReg = 0;            /* Asserts remote event */
        Wait(&temp);                 /* now wait to be BPSignalled */
        /* TAS cleared in the remote processor's event handler */
    }
}
```

Figure 6.7: Implementation of `BPRemoteWait()`

6.7 System Use and Configuration

The source code is first configured for the required platform using a set of `sed` scripts that generate the `make` files appropriately. Having configured the installation, the whole system is compiled from a single `Makefile` in the top level directory. The following discussion concentrates on the Helios version of Grape although most of it applies equally to the system running on any platform.

Before Grape is started, the backplane event handlers must be installed on each processor. A simple installation program is needed to set up the event handler (for the benefit of Helios) which eventually forks the handler shown earlier. A simple shell script is

```
void BPEventHandler(word *Data)
{
    Semaphore *s = (Semaphore*)pr->EventReg;
                                /* this read clears event */
    if( s != NULL ) {           /* was it a BPSignal or a BPRemoteWait */
        Signal(s);
    } else {
        ProcRegs *pr = (ProcRegs *)Data;
        Semaphore *local = (Semaphore*)pr->localSem;
        Semaphore *remote = (Semaphore*)pr->remoteSem;
        pr->TASFlag = 0;        /* now release ourselves */
        Fork(512,BPremWait,2*sizeof(Semaphore*),local,remote);
    }
}

/* The forked process that must perform the local wait */
static void BPremWait(Semaphore *local, Semaphore *remote)
{
    Wait(local);
    BPSignal(remote);
}
```

Figure 6.8: The Backplane Event Handler

used to carry out this procedure for each processor in the system.

Much of the final configuration of the system is controlled by a set of start-up files. The user runs the program `sysload` from the command line which is responsible for loading the remainder of the system. `sysload` reads in configuration details from a file called `sysload.rc` unless another filename is given as the only command-line argument. `sysload.rc` contains a number of lines of the form

```
<processor> <task> [<argument> ...]
```

followed by a terminating * on a line by itself and a set of connection statements that `sysload` uses to interconnect the processors. Typically, the system loader will run a copy of the dispatcher on each processor and fully interconnect the network.

The interconnection process involves using message passing to distribute the shared memory locations of shared structures such as the dispatcher's global work queues and the fact base access table. In the case of a small four processor system, `sysload.rc` may look like

```
/00 dispatch init init1.rc
/01 dispatch init init2.rc
/02 dispatch init init3.rc
/03 dispatch init init4.rc
*
connect-all
```

In addition to loading the system modules and initialising the global fact base access table, the system loader also runs a command-line interpreter process in a separate window that is used to interrogate the system about current system conditions. The CLI implements a very limited set of commands but allows tasks to be started manually and allows certain system parameters to be monitored.

The dispatcher is run with the argument list of a single process on its command line. The dispatcher will initially interact with the system loader while shared memory addresses are being distributed and then it will begin work by spawning its first process. Typically the first process to run will be responsible for starting other processes on that processor. In the example above, the dispatcher will initially run a task called `init` and pass it the remaining command line arguments (the `init?.rc` files).

The `init` task, in the case of Grape, is able to preload (or set as cached) fact bases as instructed in its input file. It then spawns a set of processes that are defined in the input file by lines of the form

```
<priority> <startTime> <compTime> <deadline> <task> [<arg1> ...]
```

If the timing information is not known or is irrelevant to this task, the appropriate fields are set to -1. Each task is spawned as requested and then the `init` task terminates. Typically the `init.rc` file will contain a number of inference engines that are to be started along with any auxiliary processes required on that node. A typical `init.rc` file would be

```
common.fb      1    1  # Preload and cache
monitor.fb     0    1  # Cache only

100        0  100  300  kbs common.rb
100        0   -1   -1  datacq datacq.rb
 50      100    3  200  kbs monitor.rb
```

The inference engines are invoked (typically from lines in an `init.rc` file) with the command `kbs <rulebase>`. The rule base contains the names of the fact bases to which it requires access. These fact base names are used as an index into the central fact base table which shows which fact bases are currently loaded and which are not. If the fact bases are already loaded they can be simply attached to, otherwise the base is loaded. The rules are then read in from the rule base and each clause is linked with the appropriate entries in the fact bases. This process allows the inference engines to operate very quickly as no searching of the fact bases is required after the initial loading phase. The inference engines proceed as dictated by the rule base.

6.8 Rule and Fact Base Compilers

The two knowledge compilers are named `rbc` and `fbc`. These programs are in fact scripts that pre-process the source code using `cpp`, the C pre-processor, and pipe the output to the appropriate compiler. Using the C pre-processor has a number

of advantages. Firstly the knowledge source files can be commented using C style comments and these will be stripped by the pre-processor but more importantly, the functionality of the macro processing can be used. This allows the rule bases to define macros, conditional compilation etc. without these having to be provided explicitly by the compilers. The '`#line`' directives generated by the C pre-processor are understood by the compilers so error messages generated will still refer to the original source file.

The fact bases should be compiled first. `fbc` will check the syntax of the fact source file and check the typing and usage of each fact definition and the instances of these facts. Provided there are no errors, the fact base will be generated ready for use. The rule base compiler must refer to the compiled fact bases as it must know the object definitions to which the rules refer and ensure that the facts referred to exist and are of the correct type. The rule source code will have a number of '`include <factbase'>`' line that instruct the compiler which fact bases are referred to by the rules. Again, the rules are checked for correctness and provided there are no errors, the rule base is output. Both compilers produce verbose error messages when they are unable to successfully compile the source code. Once an error is encountered, the compilers will continue to the end of the source file but will not generate any output. Continuing after the first error often allows a number of simple errors to be corrected in one compile/edit iteration but, as with any compiler, some errors may cause spurious errors to be found in the following source code. Some warnings are also generated by the compilers that should not be ignored but do not cause the compilation to be abandoned. Typical warnings are the assignment of a floating point fact to one declared as an integer.

6.9 Summary

This chapter has dealt with the implementation details of the Grape system. Much of the chapter has referred directly to the Helios version of the code but the basic principles are identical for the system running on any platform. The roles of the system dispatchers and the agenda managers was described and the manner in which they communicate. The implementation of the co-operative scheduling scheme and the remote semaphore operations was also presented. The following chapter goes on to show how well the system performs in practice.

Chapter 7
The Performance of Grape

The two previous chapters have presented the design and implementation details of Grape. In this chapter, the performance results of Grape are presented. The local and global scheduling policy and the knowledge-based system elements of Grape are all considered.

7.1 The Local Scheduler

7.1.1 Co-operative Scheduling

Providing a co-operative scheduler has enabled a real-time scheduling policy to be implemented on systems that could otherwise not support a real-time system. This is achieved at the cost of some overhead and a reduction in system sensitivity but the implementation of the co-operative scheduler in Grape, with the inclusion of the I/O optimisation has minimised these costs.

The average time taken for a typical context switch including switching from the running task to the dispatcher, deciding which task to run next and scheduling that task is shown in Table 7.1. The three figures are all for T800 systems but the first is using a standard PC plug-in card, the second is a normal processor in the Bath transputer system and the final figure is for the T800 processor running on the VSC card of the

Processor Node	Time (μs)
TRAM T800	85.35
Rack Processor	107.26
VSC Processor	127.83

Table 7.1: Typical Context Switch Times

Bath system. The difference in these times is caused by the speeds of the memory interfaces. It must be remembered that much of this overhead would be incurred by a real preemptive scheduler as two task switches plus some computation must still be performed.

Scheduling efficiency depends, as in any system, on the rate at which the scheduler is called. Clearly, if the co-operating process runs for only a few micro-seconds before surrendering the processor then the system efficiency will be low. In practice, if the scheduling grain can be maintained at around 10ms then the total overhead for the scheduler including the dispatcher, and the local and global schedulers is typically less than 1% of the total run-time. In the worst case, the response time of the system to some event which should cause a new process to be scheduled will also be 10ms if the process has only just been scheduled when the event occurs. In practice the mean delay will be half the scheduling granularity but if this proves to be unacceptable for a particular application, the developer must make some compromise between efficiency and responsiveness.

7.1.2 The STL Algorithm

The STL algorithm was designed to provide a fast guarantee algorithm. When assessing the speed of the STL, a random set of (non-prioritised) processes was generated and used as the input to the STL scheduler and an optimal scheduler based on the earliest deadline algorithm. On average the STL can guarantee a task in about 1–5% of the time taken by the implementation of the earliest deadline. Each call to the STL algorithm takes between 200 and 600μs compared to between 6000 and 45000μs for the earliest deadline. One of the advantages of the STL is that under heavy loads, the STL reduces in size (because less spare time exists) therefore increasing its speed. The work performed by the earliest deadline algorithm increases with processor load leading to slower guarantee speeds at the very point where it is needed to be at its most efficient.

The STL algorithm is sub-optimal and occasionally it will falsely reject a task that it could have scheduled. This occurs because the algorithm is forced to commit the schedule slack to a particular location in the STL. This effect can occur at any level of processor load but in practice, the failure rate is very low for low to medium processor usage. Typical graphs of STL performance are shown in Figures 7.1–7.8, included at the end of this chapter. These graphs were generated using random task sets with different mean computation times and laxities. The failure rate is the percentage of tasks rejected that were successfully scheduled by an ideal earliest deadline scheme that takes zero time to execute. For these tests, schedules of 1000 time units were used.

The failure rate does become significant for higher processor loads but the guarantee overhead is of the order of 100 times lower than an optimal scheme. The significant

overhead of an optimal guarantee algorithm does in general cause a greater number of task failures than the STL. The speed of the STL also greatly improves the global scheduling performance and a large proportion of these rejected tasks will be scheduled successfully elsewhere.

7.2 The `procgen` Program

`procgen` was written to generate test data for experimentation with the global scheduler. It takes a configuration file as input which describes the basic characteristics required of the output task set. These include the number of processors; overall schedule length; minimum, average and maximum task length; average laxity etc. The strictness of the schedule can also be specified along with the overall processor load required. All this data is used by `procgen` to generate a schedule, in memory, of the specified characteristics. As its output `procgen` produces a list of tasks including the start-time, computation time and deadline of each one. This task list can then form the input to the algorithms under test.

All tests on the global scheduler were performed in this way. The task list was read in by another process running under the dispatcher on each processor that spawned tasks with the stated characteristics. Each task was spawned at a random time between the start of the experiment and the process's start-time.

When testing the global scheduling algorithms, task sets were generated for each processor in the system. Task sets of varying loads were generated for each of the processors except one which received an over full task set (ie. one that could not be completed on a single processor).

7.3 The Global Scheduler

The global scheduler was implemented with four different algorithms: no migration; random migration; focussed addressing and bidding. For testing purposes, each algorithm was assessed in two states: single migration and multiple migration. In the single migration case, when a process is migrated to another processor which in turn finds it cannot schedule the process, it is rejected. With multiple migration, the receiving process is allowed to go on to migrate the task again to some other processor. The single migration case gives an indication of the accuracy of the algorithm's assessment of processor loads although the multiple migration system would be used in reality. First each algorithm is discussed and then results of each algorithm under a variety of system loads is discussed.

7.3.1 No Migration and Random Migration

The no migration scheme is a simple control case included as a baseline from which to compare the other algorithms. It will reject any process that it cannot guarantee locally. The random scheduler chooses to migrate the task to a random processor. Although the random scheme has (by definition) no knowledge about the other processors in the system the scheme works reasonable well as will be seen later. A refinement of the random scheme is possible when multiple migrations are allowed. In this scheme the nodes already visited are stored in the migration message so any further migrations will not try processors that have already rejected the task.

7.3.2 The Focussed Addressing Scheme

The focussed addressing algorithm relies on data broadcast periodically by each processor concerning their current load state. After each call to the STL a processor broadcasts two numbers to every other processor in the system. These numbers are the `time` field and the `slack_total` field of the head element of the STL. These two numbers are used to give a rough guide to processor loads.

For accurate migration decisions, entire STL's would have to be transmitted and evaluated. This focussed addressing scheme uses only a fraction of that information on which to base its judgement. The load data will often be out of date and clearly, the two numbers give no indication of the position of the slack time but as an heuristic migration policy, this performs very well.

When a processor is unable to guarantee a task locally, it uses the load information of each processor to determine where to migrate the task. This evaluation involves calculating a factor f for each processor p. If $d_i > L_p$ then f_p is given by

$$f_p = (d_i - L_p) + S_p$$

otherwise,

$$f_p = S_p$$

where S_p is the `total_slack` and L_p is the last `time` value of processor p. A migration candidate is chosen by selecting the processor with the highest value of f_p.

Clearly, if $\max(f_p) < c_i$ the process cannot be migrated and is rejected but otherwise it is migrated to processor p. To improve efficiency slightly, a threshold value

FA_THRESHOLD is used. If any value of f_p is greater than FA_THRESHOLD then the task is migrated immediately without having to calculate the remaining f_p values.

7.3.3 The Bidding Scheme

The bidding algorithm is the most complex of those implemented. It works by sending bid requests, to every processor, containing the characteristics of the task it is unable to guarantee. Each dispatcher will receive this bid request and decide whether or not to make an offer for the task. This decision will depend on the current workload (which may prevent the request even being considered) and the result of an investigation of the current STL. The original processor must receive the offers from each processor that replies and decide which processor shall be granted the task. There are many parameters involved with the bidding algorithm each of which has an effect on the overall performance of the scheme.

The processor which makes the initial request must set a time by which the bids must have arrived. This time out is set to be the task's deadline minus the parameter BID_SLACK which must reflect the maximum amount of time required for the task to be migrated to and scheduled on the remote processor. This parameter has been set empirically and is highly system dependent. As offers arrive from remote processors the best one is maintained. When the dispatcher notices that a bid has timed out, it will migrate the task to the processor which has made the best offer.

A processor making a bid does not know if it will be granted the task or not so it would be very inefficient for the processor to actually reserve this space. Instead when migrated tasks are received they are subjected to the guarantee algorithm once more

and may be rejected or re-migrated at that point.

This procedure can be made more efficient. If a very high bid is received from some processor there is little point waiting for further offers. Each bid is compared with the second parameter `BID_THRESHOLD` and if it is greater the task is migrated immediately.

The remote processors must decide whether or not to make a bid based on the current state of their STL's. It is possible to call the STL specifically for this purpose and it will return a floating point number equal to the total number of processes similar to this one that it could currently guarantee. Clearly, if this number is less that 1, the task cannot be scheduled and a bid is not made. If however, the number is only slightly above 1 the dispatcher must decide whether or not to make a bid. With a moderate number of processors, it is reasonable to expect a number of bids to be made for each task. If that is the case, the bidding traffic can become significant and the input work queue of the processor initiating the bidding will become loaded with offers. To overcome these problems an `OFFER_THRESHOLD` parameter is used to filter out the marginal bids.

When multiple migration is allowed, the processor which receives the task initially may find that it is unable to guarantee it. If this is the case, it is clearly inefficient for a new auction to take place when the information was only recently obtained. To overcome this each task migration includes within it the processor number of the second best offer (if one existed). This number is then used directly by the second processor. Other solutions to this re-migration policy could involve switching to a focussed addressing scheme for subsequent migrations or returning the task to the sending process which may by then have received new bids.

7.3.4 Comparing the algorithms

Tests were performed using each algorithm as described earlier. All processors were loaded with a set of tasks to the appropriate load level and one processor was given a work load that was unschedulable on a single node. The total number of task failures (deadline misses) was recorded for each algorithm. It should be noted that the optimal solution to these schedules should not necessarily be 0 failures because the excess tasks on processor 0 may coincide with tasks on the other processors. Each algorithm has a number of parameters and these may be set using command line options to the dispatcher. The algorithm used is also determined in this way. Table 7.2 shows the results obtained from a large number of runs. Each figure is an average of 5 different experiments. Figures are presented for the system running on six and twelve processors and with between 20% and 80% load on the other processors. The mixed load case has processors with various loads between the two extremes. The algorithm names require some explanation (the names shown are the command line options given to the dispatcher). '`-n`' indicate no migration, '`-r`' is random and '`-R`' is random without duplications. The '`-m`' in all cases refers to multiple migrations—when it is not present, a task could only be migrated once. The bidding algorithm is specified as '`-B<BID_THRESHOLD> -O<OFFER_THRESHOLD>`' and the focussed addressing scheme as '`-F<FA_THRESHOLD>`'.

There is a great deal of data represented in Table 7.2. The key points to be raised from it are summarised below.

- Every algorithm does better than the no migration control case in all test cases.
- The random, single migration algorithm (`-r`) performs badly and is only better

Total	20% load				50% load				80% load				Mixed load				Migration
	6 Procs		12 Procs		6 Procs		12 Procs		6 Procs		12 Procs		6 Procs		12 Procs		Algorithm
106.67	5.67	(1)	7.33	(1)	13.00	(3)	16.67	(4)	19.33	(9)	25.67	(2)	9.00	(1)	10.00	(1)	-F1.0 -m
107.33	6.33	(4)	7.67	(2)	13.33	(4)	15.67	(3)	19.33	(9)	23.67	(1)	10.00	(8)	11.33	(6)	-R -m
111.01	5.67	(1)	8.00	(4)	12.67	(1)	15.33	(1)	21.67	(21)	28.00	(5)	9.67	(4)	10.00	(1)	-F0.0 -m
112.67	6.00	(3)	7.67	(2)	14.00	(7)	19.00	(6)	21.00	(14)	26.00	(3)	9.00	(1)	10.00	(1)	-F2.0 -m
115.01	8.67	(24)	8.00	(4)	12.67	(1)	15.33	(1)	18.67	(2)	26.67	(4)	13.00	(23)	12.00	(8)	-r -m
120.34	6.33	(4)	10.00	(10)	15.00	(17)	18.00	(5)	20.67	(13)	29.67	(10)	10.00	(8)	10.67	(4)	-F0.0
120.66	6.33	(4)	9.33	(8)	13.33	(4)	20.33	(10)	20.67	(13)	30.33	(11)	9.67	(4)	10.67	(4)	-F1.0
123.66	6.67	(7)	11.00	(20)	14.33	(12)	21.00	(14)	21.00	(14)	28.33	(6)	9.00	(1)	12.33	(9)	-F3.0 -m
123.68	7.00	(10)	10.67	(15)	13.67	(6)	20.67	(13)	17.67	(1)	30.33	(11)	10.00	(8)	13.67	(16)	-B2.0 -O0.5 -m
124.68	6.67	(7)	9.00	(7)	14.67	(14)	21.67	(15)	21.00	(14)	29.33	(9)	10.67	(12)	11.67	(7)	-F2.0
125.34	7.00	(10)	10.33	(11)	14.00	(7)	20.33	(10)	18.67	(2)	31.67	(16)	10.67	(12)	12.67	(12)	-B2.0 -O1.0 -m
127.00	6.67	(7)	9.67	(9)	15.33	(21)	22.00	(18)	21.00	(14)	29.00	(8)	11.00	(16)	12.33	(9)	-F3.0
127.33	7.00	(10)	10.67	(15)	14.33	(12)	19.67	(7)	18.67	(2)	32.33	(19)	10.33	(11)	14.33	(19)	-B2.0 -O0.5
127.34	7.00	(10)	10.67	(15)	14.00	(7)	20.00	(8)	19.00	(6)	31.67	(16)	11.00	(16)	14.00	(18)	-B2.0 -O1.0
129.00	7.33	(16)	10.33	(11)	15.00	(17)	21.67	(15)	21.67	(21)	28.33	(6)	11.67	(19)	13.00	(13)	-F4.0 -m
129.68	7.33	(16)	11.00	(20)	14.67	(14)	21.67	(15)	18.67	(2)	31.00	(14)	9.67	(4)	15.67	(22)	-B3.0 -O1.0 -m
131.66	7.33	(16)	11.00	(20)	15.33	(21)	22.00	(18)	19.00	(6)	32.00	(18)	10.67	(12)	14.33	(19)	-B2.0 -O1.5 -m
132.34	7.67	(21)	11.00	(20)	15.00	(17)	22.67	(21)	19.33	(9)	33.67	(23)	10.67	(12)	12.33	(9)	-B2.0 -O1.5
133.65	7.33	(16)	10.33	(11)	16.33	(24)	23.33	(24)	19.00	(6)	32.67	(20)	11.33	(18)	13.33	(15)	-B2.0 -O2.0 -m
133.66	8.00	(22)	10.33	(11)	15.33	(21)	23.00	(22)	21.67	(21)	30.33	(11)	12.00	(20)	13.00	(13)	-F4.0
134.01	7.33	(16)	10.67	(15)	14.00	(7)	23.00	(22)	20.67	(13)	33.67	(23)	9.67	(4)	15.00	(21)	-B3.0 -O1.0
135.02	7.17	(15)	8.84	(6)	15.00	(17)	20.17	(9)	20.33	(12)	32.84	(22)	12.50	(22)	18.17	(25)	-r
135.67	7.00	(10)	10.67	(15)	16.00	(24)	23.67	(25)	21.33	(20)	31.33	(15)	12.00	(20)	13.67	(16)	-B2.0 -O2.0
141.33	8.33	(23)	11.00	(20)	14.67	(14)	20.33	(10)	22.00	(24)	32.67	(20)	16.00	(25)	16.33	(23)	-B1.0 -O1.0 -m
146.00	9.00	(25)	11.33	(25)	14.00	(7)	22.00	(18)	22.33	(25)	35.67	(25)	14.67	(24)	17.00	(24)	-B1.0 -O1.0
206.33	17.33	(26)	19.67	(26)	22.33	(26)	30.67	(26)	25.33	(26)	41.67	(26)	21.33	(26)	28.00	(26)	-n

Table 7.2: Global scheduling results showing the average task failures for each algorithm. The '-n' row is for no migration. The four main columns are for different load averages on the non-overloaded processors. Within each column are results for 6 and 12 processor cases. The numbers in brackets are ranks for each test. The first column is the total number of failures for each algorithm and the table is sorted on this number.

than poorly tuned algorithms. The only test in which it proves reasonable is with low processor loads in the 12 processor tests. This is the easiest scenario for the schedulers so this is not a significant achievement.

- The focussed addressing algorithm performs well in general. Clearly higher values of the `FA_THRESHOLD` are worse suggesting that the heuristic of offering immediately to a high bid is unsound. This is understandable given that the load information is inherently inaccurate. It appears however that the optimal threshold is non-zero as demonstrated by the performance of '`-F1.0`'.

- The '`-F1.0`' performs very well in most cases but is let down by its poor performance for 80% load on 6 processors. In this case there is very little room for error and the inaccuracy of the load information is shown up.

- The bidding algorithm generally performs worse than the focussed addressing scheme. This is caused by the overhead of the algorithm given the speed of the backplane. The focussed addressing scheme is able to migrate wrongly and have its mistake corrected in the time taken for a bidding auction. The exception to this is again for 80% load on 6 processors. The bidding algorithms here assess the situation much better than the other algorithms and result in less failures. For the parameters, the results suggest that a value of 2 for the `BID_THRESHOLD` and a low value of the `OFFER_THRESHOLD` seem to be best. For a bid threshold of both 1 and 4, the performance is noticeably worse.

- Re-migration affects the algorithms differently. The bidding algorithm gains very little from migration due to the accuracy of the initial migration. The random policy performs much better with re-migration as would be expected and when the option of not re-migrating to an already failed processor is used (`-R -m`), the

results are very good. Similarly the focussed addressing scheme benefits from re-migration again because mistakes can be corrected.

- The mixed load case is interesting for a number of reasons. It is the most realistic system load (albeit not the most desirable for efficiency) and the results for these cases are slightly different. The random algorithms do much worse under these conditions because they have no means of avoiding the few heavily loaded processors. The focussed algorithms again prove best. For the 6 processor case, the bidding algorithm with a high bid threshold also performs well. However, because of the algorithms broadcast approach, the good performance does not seem to scale to the 12 node case.

The results show that in general a focussed addressing scheme performs better than the alternative algorithms for this system. Its performance in the mixed load situations in which most real world applications will operate is also encouraging. However, the bidding algorithm appears to be useful especially when the processors are heavily loaded. It may be possible to combine these algorithms into a new switching algorithm that adopts the correct behaviour under all circumstances but this has not been investigated. The results are individual to the Bath transputer machine but show important trends that would be true for any system.

The Grape system is obviously able to use all of these algorithms. The focussed addressing scheme with a `FA_THRESHOLD` of 1.0 would be the recommended (and default) algorithm initially but each application really should be benchmarked in this way to determine the best policy for their particular load patterns.

7.4 Reasoning Performance

The speed of the inference engine will clearly affect the overall system performance. The time taken for a single inference is a good measure of the inference engine speed but this will vary considerably depending on the complexity of the rule and whether or not the rule is fired. The results shown in Table 7.3 give average inference times for a set of rules of differing complexity. Some of these test rules are from the engine diagnostic system whereas others are artificial test rules. The rules have varying numbers of antecedents and actions. The complexity of the clauses also varies and some rules fire while others do not. Clearly, there is a wide variance in these results and there is no upper limit as rules of arbitrary complexity can be written. Executing a single rule can take as little as $89\mu s$ for a rule containing a simple comparison that fails. The same rule takes nearly $154\mu s$ if the condition is true and a single action is performed. The rule that requires $1300\mu s$ in these figures performs variable instantiation and some floating point calculations. Clearly the time required will be application dependent but these figures give an indication about the speed of the inference engines.

The basic inference speed is very fast. All the results given in Table 7.3 are for the KBS running on a 20MHz T800. As Grape is highly portable it has been possible to compile the system on a wide range of platforms, many of which are included in Table 7.4. These figures have been obtained using the same rules as the T800 test above. The average figure is somewhat meaningless in this context so a simple range is given. Clearly, some processors will perform better than others on integer calculations, others on floating point etc. As the goal of the exercise was not to benchmark these systems but merely to get some idea of how Grape performs on them these effects are not analysed.

Rule	Time (μs)	Inferences/second
1	89.56	11 166
2	153.84	6 500
3	405.72	2 464
4	89.56	11 165
5	163.51	6 115
6	541.24	1 847
7	105.20	9 505
8	100.65	9 935
9	901.36	1 109
10	243.92	4 099
11	1361.93	734
Average	377.96	2 646

Table 7.3: Inference Speeds

From a developer's view, it is also important that the knowledge base development cycle is as efficient as possible. The rule base and the fact base compilers are fast (due to the simplicity of the languages) and error messages are verbose. It is usually very clear where syntactical errors have been made allowing the knowledge engineer to concentrate on the content of the knowledge bases. Some debugging information is available from the inference engines that will show each inference step and why it was performed. This level of debugging can often prove very useful.

Assessing Grape's performance in parallel applications is very difficult. As Grape itself is a programmable expert system, its speed and efficiency, like any other parallel system, depend on the input program. If each processor run independent rule bases then a linear speedup is possible. In more realistic systems, something less than this will be achieved. Grape itself is quite fast and light weight so well balanced applications with good

Processor	Operating System	Rule Times (μs)	
		Minimum	Maximum
SGI Indigo	IRIX 4.0.5F	4.33	55.58
Sun Sparc 10	SunOS 4.1	4.67	63.91
Sun Sparc 10	SunOS 5.2	6.92	84.08
Sun Sparc II	SunOS 4.1	7.08	117.50
486DX 50MHz	MSDOS 5	7.14	86.79
Sun Sparc Classic	SunOS 5.2	12.08	214.24
Sun Sparc IPC	SunOS 4.1	12.42	216.24
486DX 50MHz	Linux	13.50	181.10
386DX/387 33MHz	MSDOS 5	29.11	317.50
386DX/387 25MHz	MSDOS 5	49.44	530.08
386DX/387 25MHz (no cache)	MSDOS 5	50.54	546.01
T800 20 MHz	Helios	89.56	1362.21
Sun 3/60	SunOS 4.1	91.66	1439.94
386SX/387SX 20MHz	MSDOS 5	101.07	1093.68
T800 20 MHz	Meiko	106.46	1519.36

Table 7.4: Inference Speed on various platforms

locality and modest inter-processor communication will perform well. Grape is unable to perform well if the input program is inefficient, badly balanced or communications bound.

7.5 Summary

This chapter has shown the basic performance of the Grape system as implemented on the Bath transputer system. The performance of the schedulers, both local and global have been described in detail and the performance of various migration algorithms were

compared. All of the implemented features of Grape work as intended. The remote fact base accessing protocol works well and incurs a very low overhead. Individual inference engines are able to spawn new processes and have control over existing ones. Although the speed of many of these features are difficult to quantify, the general performance and responsiveness of the system is very good.

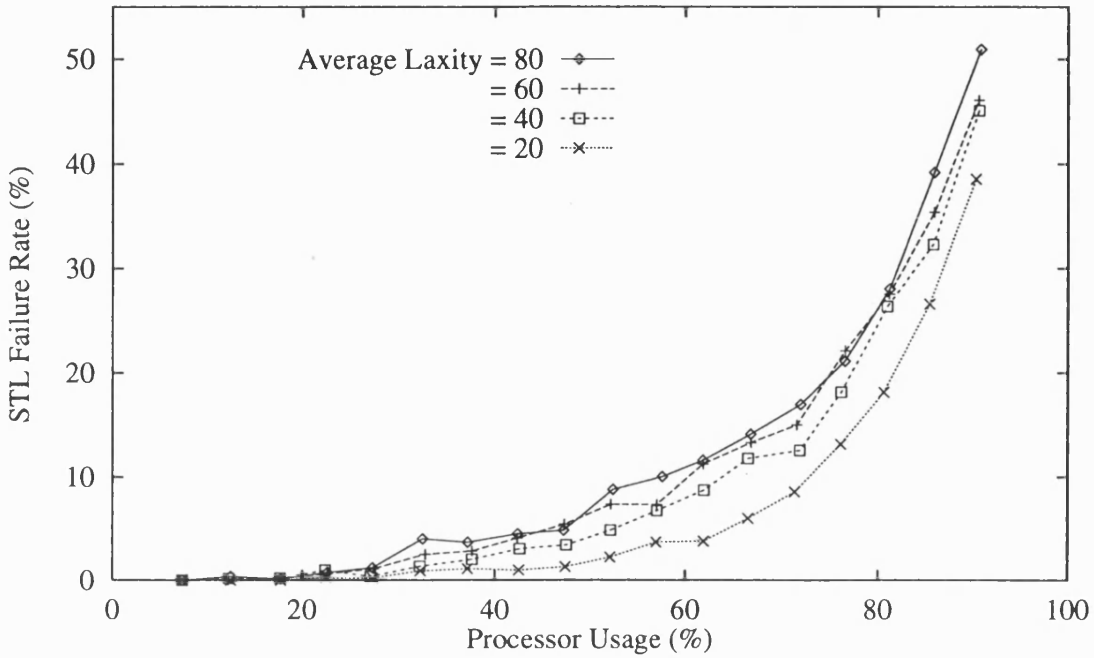

Figure 7.1: STL Failure Rate. ($\overline{c_i} = 20$)

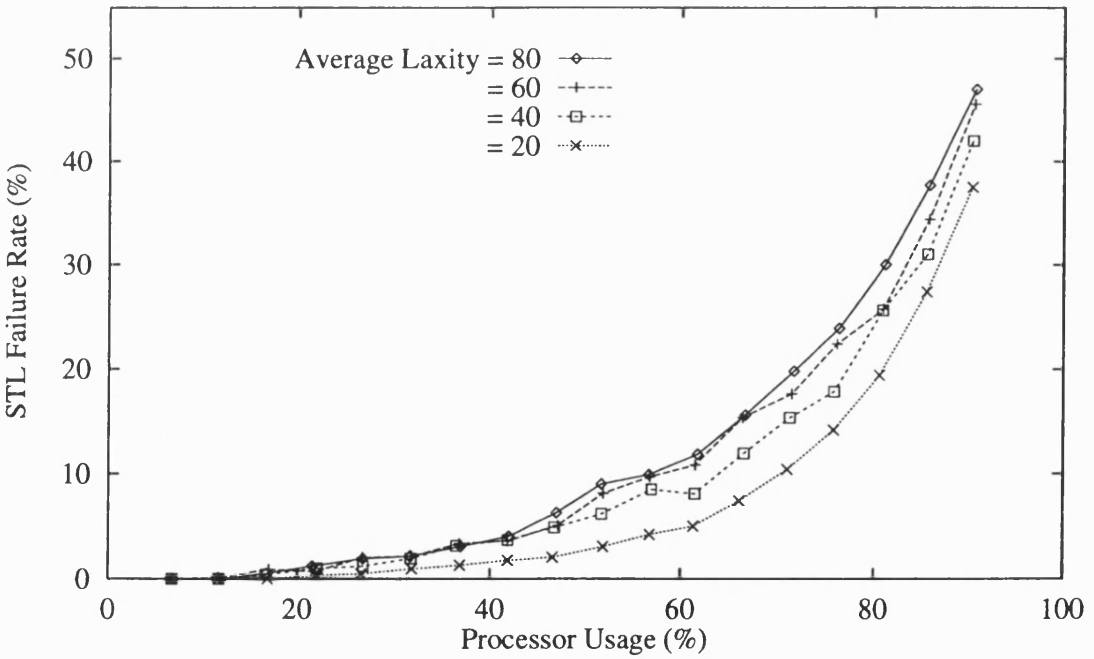

Figure 7.2: STL Failure Rate. ($\overline{c_i} = 40$)

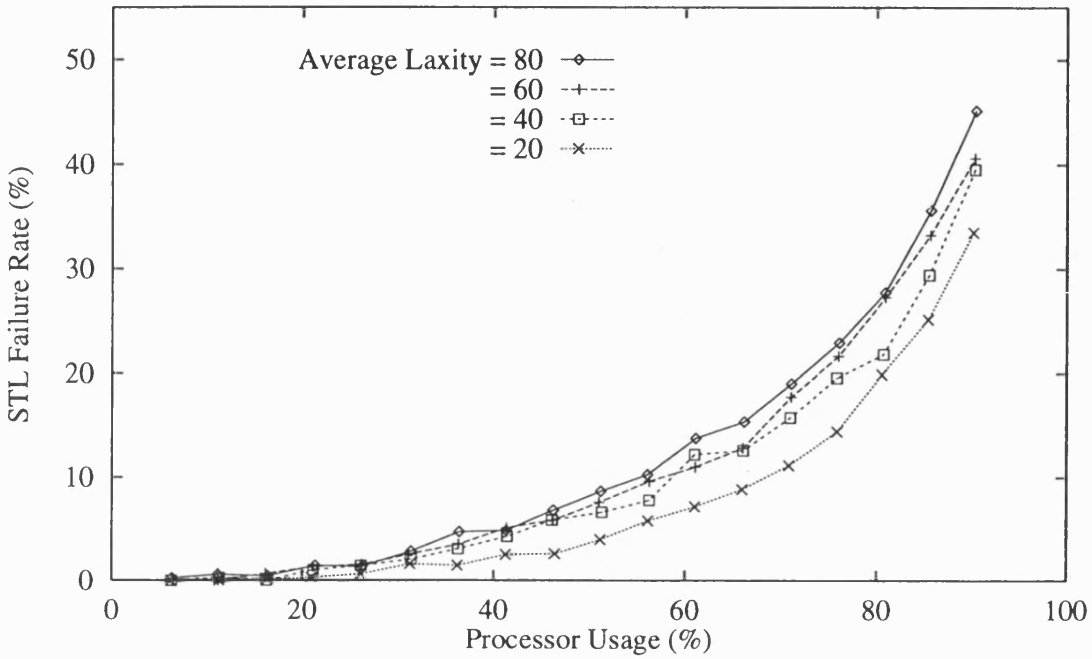

Figure 7.3: STL Failure Rate. ($\overline{c_i}$ = 60)

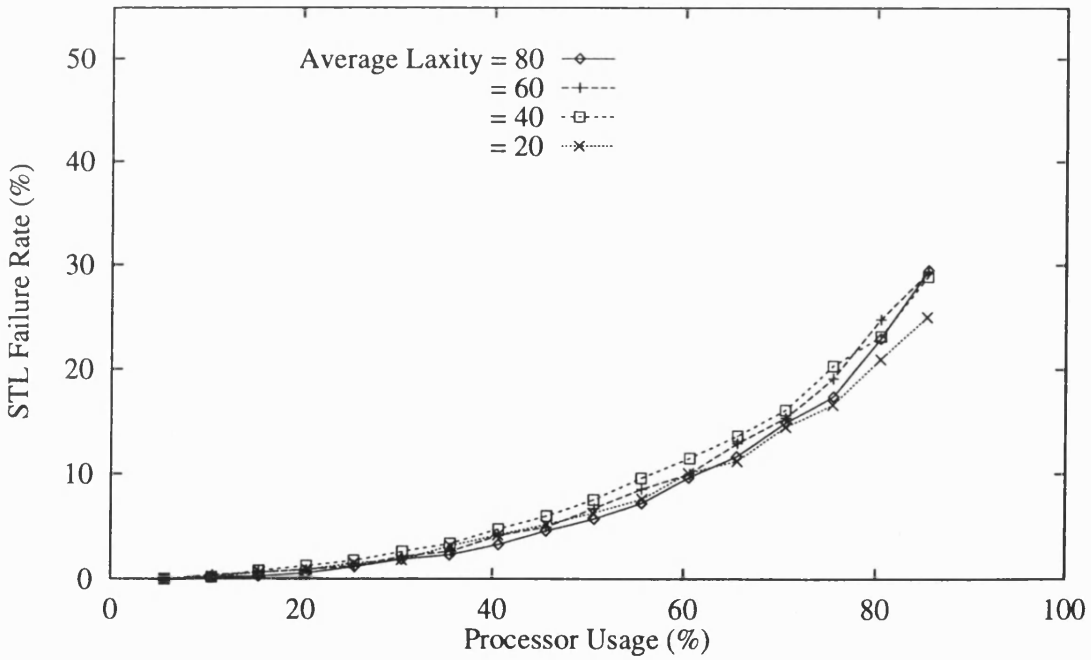

Figure 7.4: STL Failure Rate. ($\overline{c_i}$ = 80)

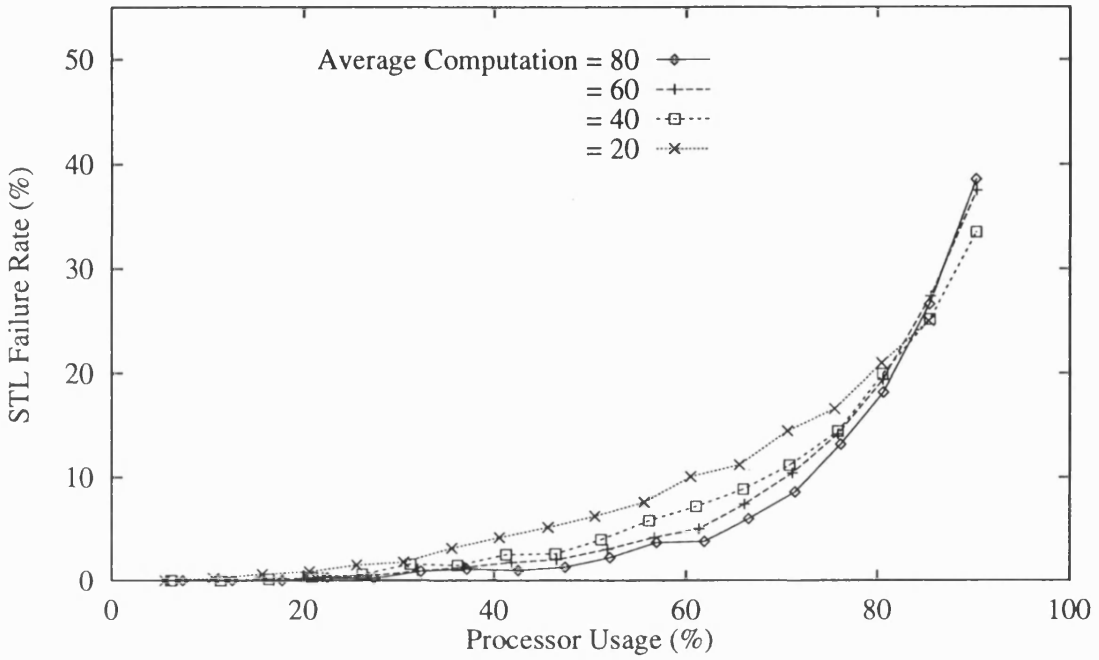

Figure 7.5: STL Failure Rate. ($\overline{l_i} = 20$)

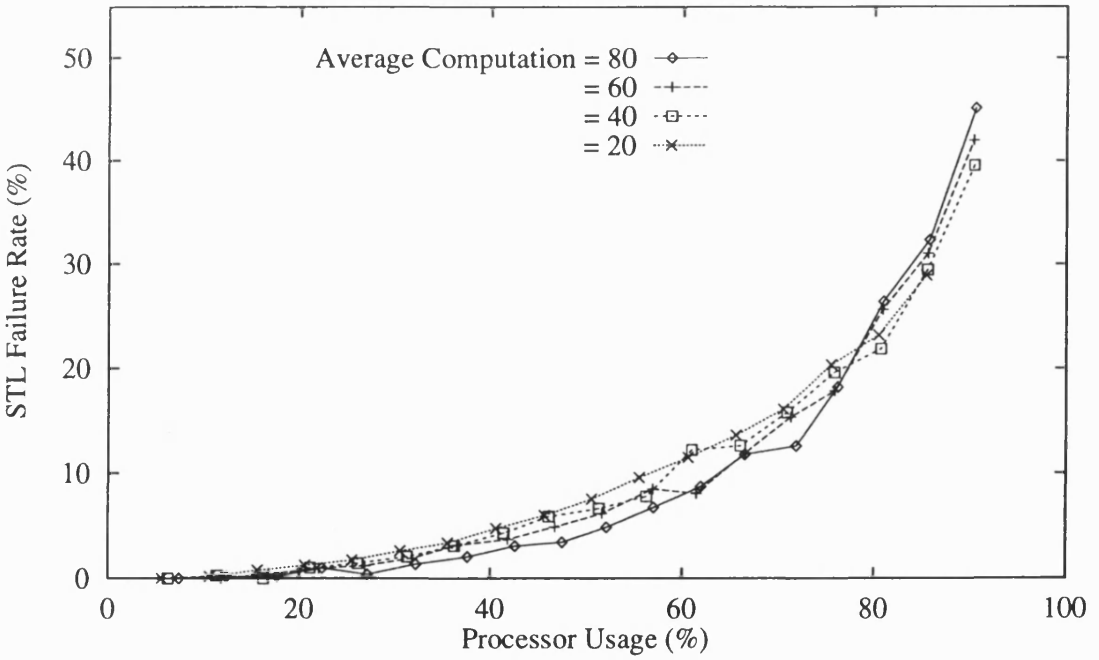

Figure 7.6: STL Failure Rate. ($\overline{l_i} = 40$)

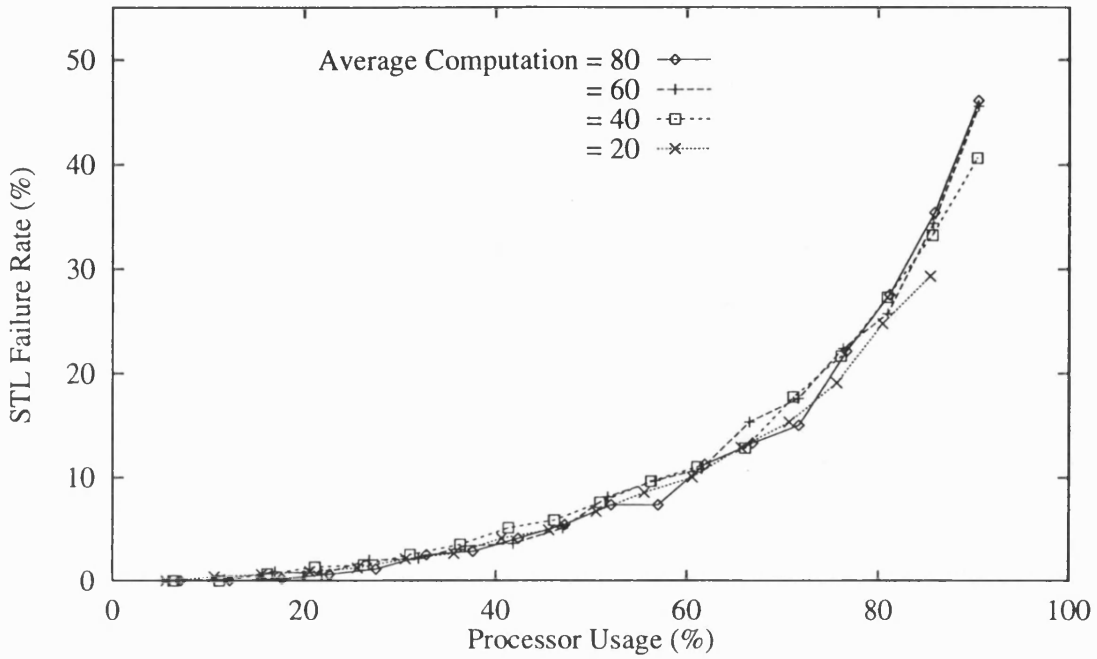

Figure 7.7: STL Failure Rate. ($\overline{l_i} = 60$)

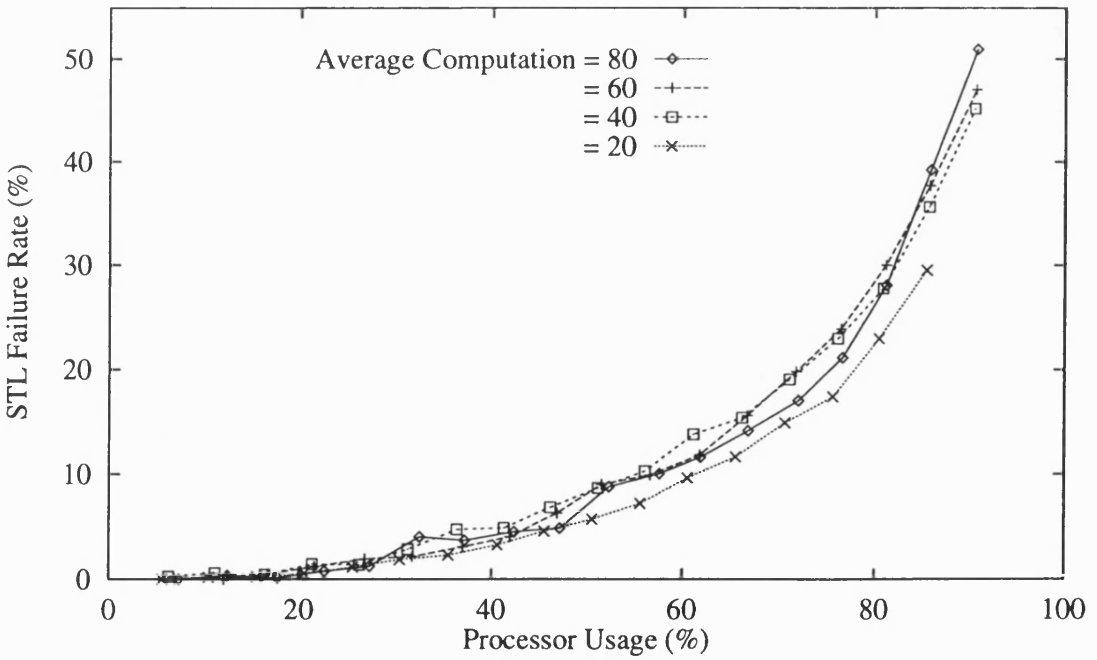

Figure 7.8: STL Failure Rate. ($\overline{l_i} = 80$)

Chapter 8

Applying Grape to Diesel Engine Diagnostics

This chapter discusses the application of Grape to a diesel engine diagnostic system (EDS). It demonstrates the use of the rule and fact base languages and shows how Grape can be used to build an EDS. Only a limited system has been built as an example but the features for a full predictive maintenance system are outlined and much of the support required has been implemented. The overall design of the EDS is presented first.

8.1 Engine Diagnostic System Design

The EDS comprises a number of modules in addition to the Grape system itself. A block diagram of the system is shown in Figure 8.1. It consists of the two data sources (the real engine and the simulator), some data acquisition and conditioning software and the Grape-based diagnostic engine. Each of these elements will be discussed individually.

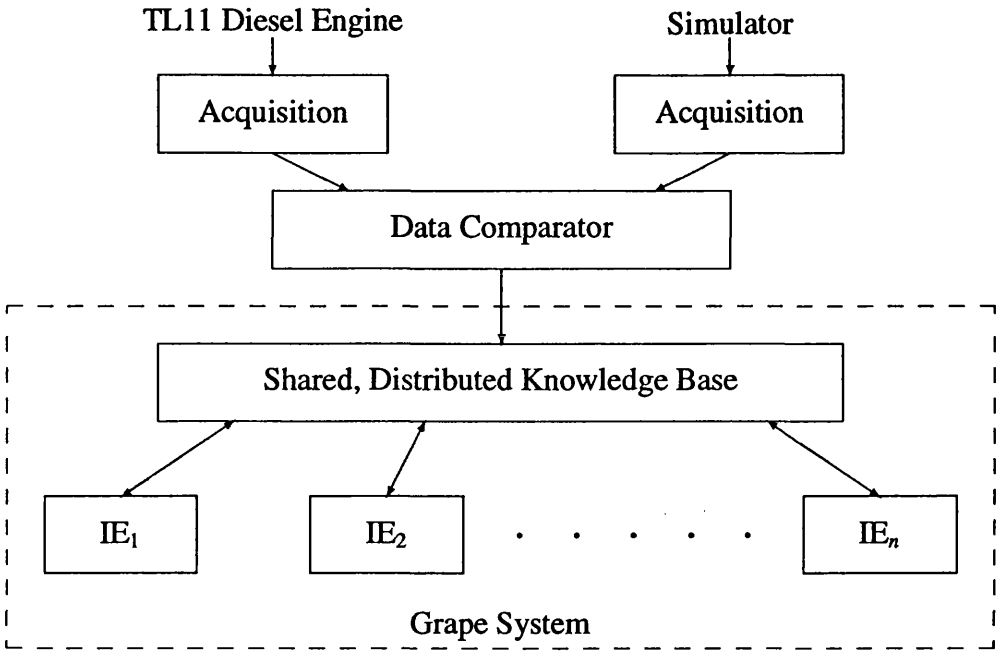

Figure 8.1: The Engine Diagnostic System

8.1.1 Using the Real-Time Simulator

The real-time diesel engine simulation is used to provide a source of comparison with the engine, and also to aid the diagnostic process. The simulator must be capable of providing data to the EDS, being controlled by the EDS and also displaying the current engine performance using its graphics capability. In normal running the simulator must track the real engine so it must be able to receive sensor values for the current operating conditions such as the speed, load, and ambient conditions.

Earlier versions of the diesel engine simulator incorporated graphical output that displayed the running of the simulation in real-time. Graphs of in-cylinder data values were presented in a form of oscilloscope display and an animated piston showed the current state of a single cylinder. These displays were very useful at the speeds at which

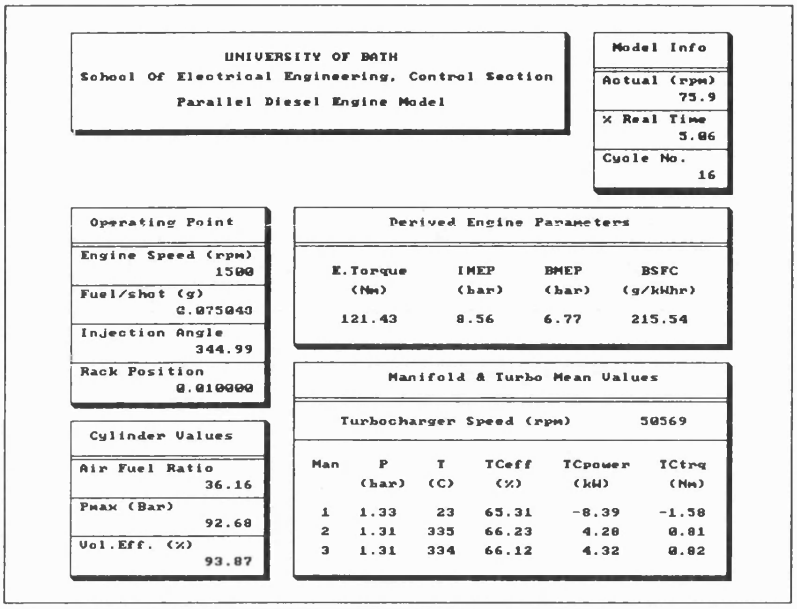

Figure 8.2: Simulator Screen: Derived Data. This is the default display that shows data derived from the raw sensor values. This data corresponds to the information an engineer would like to know in order to assess the running condition of the engine.

the simulator was then running but as the simulation speed became faster it became impossible to display these sorts of pictures.

The transputer version of the simulator relied on a text terminal as the sole on-line interface between the simulator and a user. This made the use of the simulator very difficult and greatly complicated the process of developing and debugging the simulator. As a result, a new graphical interface was designed that could display snap-shots of fast changing data, animated displays of slower derived data (such as BMEP) and plot snap-shot graphs of a wide range of engine parameters. Some sample displays from the engine simulation are shown in Figures 8.2–8.5. A consequence of this development was that the terminal interface was free to be used for user interaction.

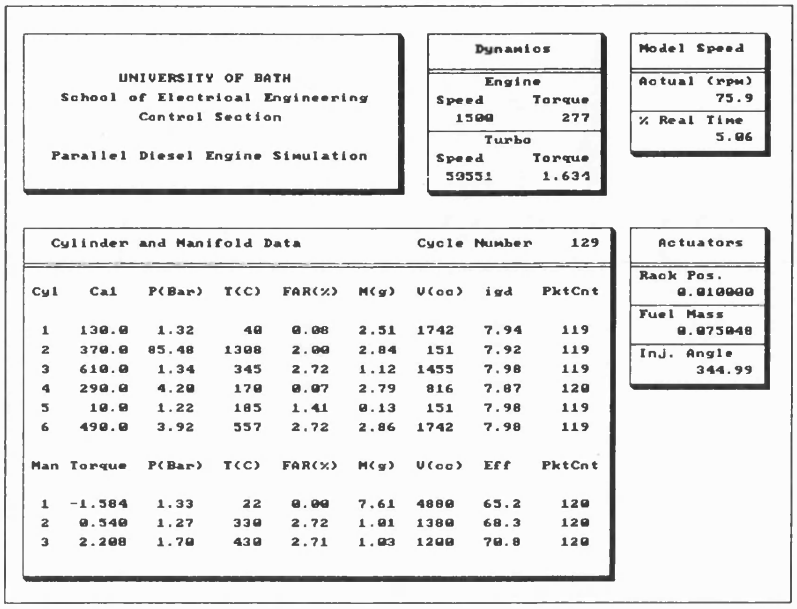

Figure 8.3: Simulator Screen: Raw Data. A fast moving screen showing raw sensor data. This screen is mainly used for debugging or monitoring particular sensors.

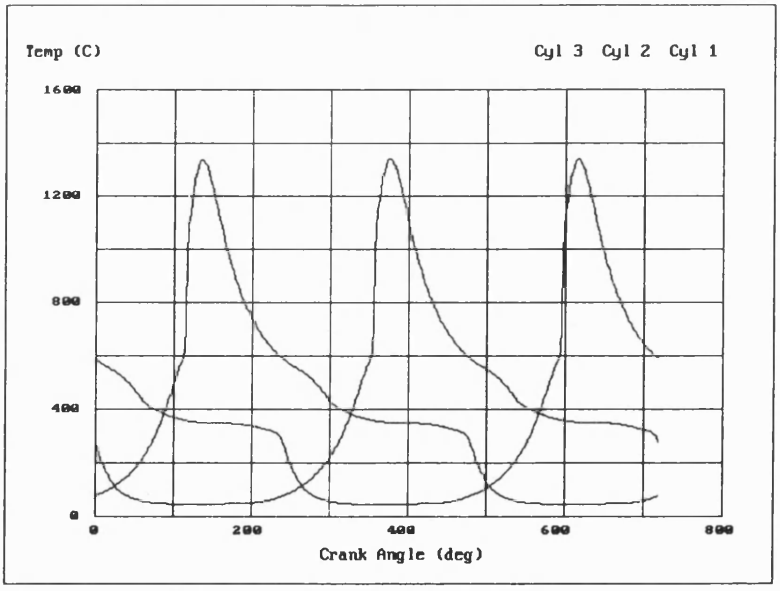

Figure 8.4: Simulator Screen: Cylinder Temperature Graphs. An example of snap-shot graphs that can be obtained of a number of simulation parameters.

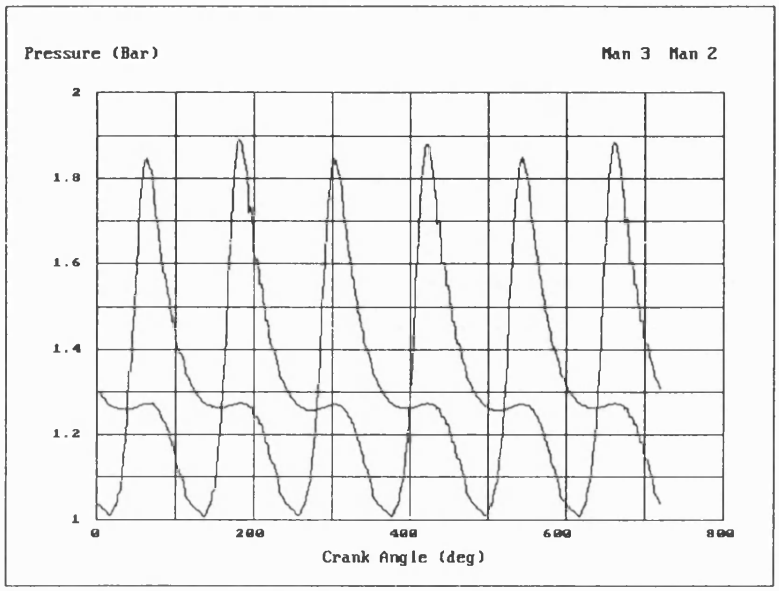

Figure 8.5: Simulator Screen: Manifold Pressure Graphs. An example of a snap-shot graph that can be obtained of a number of simulation parameters.

Another limitation of the simulator as far as this work was concerned was the lack of access by other programs running on the machine. Clearly for an on-line EDS it is important that the system can control the simulation and is able to receive information and data from it. This capability has been achieved by re-implementing the simulation as a Helios server. As with any Helios server, once created its name resides in the global name space and can be accessed by any process in the system. A simple control protocol was developed to allow clients to manipulate the simulation and obtain data from it. The interactive user interface itself was separated from the simulator and implemented as a stand alone program that simply issues commands remotely to the simulation via the General Server Protocol (GSP). A consequence of this re-implementation is that multiple clients can access the simulation simultaneously. Such concurrent access will normally not be required (or wanted) but it is possible for example to have the EDS controlling and using the simulation while a user is selecting graphics displays or logging data. Such access could easily be made 'read only'.

With the server interface, the simulator data acquisition software becomes a client of the simulator. It must simply request the required data and pass on the information to the data comparator.

8.1.2 Engine Data Acquisition

Obtaining the engine sensor data is a relatively straight forward task. A set of analogue to digital converters are required and these should be mapped into the global address space of the transputer system. Appropriate hardware was designed by Haysom [4] for the 68000 based systems so only the memory interface of his design needs to be updated for the transputer system. The software driver required needs to transfer data, as required, from the memory of the data acquisition system to the appropriate fact bases.

8.1.3 A Generic Approach to Data Acquisition

The Grape system has hooks to enable data acquisition modules to be interfaced but still requires specific source code modifications to support the functionality of each module. A better approach would be to implement a generic interface that would allow device drivers for particular data acquisition tasks to use a standard interface. The majority of each device driver should be identical. A common protocol would be implemented and the interface to the fact bases would be identical. Only the small device dependent routines would need to be written for each individual driver. In general, this interface could be bi-directional allowing actuators and other output devices to be interfaced in the same manner. The drivers would need to be parametrised to allow, for example,

sample rates and logging to be adjusted.

Each device driver should spawn a separate process to perform the I/O. This decouples the interfaces and allows data to arrive asynchronously. A buffering system should be present to allow data to be received even when the current data value is locked by one of the system inference engines. The device drivers will typically use very little CPU time and hence it is unnecessary to incorporate them into the cooperative scheduling scheme. In a system with a true preemptive scheduler, the device drivers will be of high priority to ensure that the fact base data is kept up to date.

The simulator interface and the engine data acquisition interface could both benefit from a generic scheme. If the EDS is able, for example, to shut down the engine when catastrophic fault conditions are detected, then this output process could also be via a generic device driver.

8.1.4 Data Comparator

The Data Comparator performs low level conditioning on the data received from the simulation and from the engine. It validates the data, ensuring that it is within certain limits and outputs a normalised measure of deviation between the simulation and the engine. The data sources store their information in fact base formats that are shared with the data comparator. The comparator then processes this data and stores the results in the shared fact bases of the EDS. Under fault conditions, some data may deviate by a factor of 10 whereas others only deviate by 10%. These deviations are normalised to give a "percentage of possible deviation" figure so that the output from the Data Comparator is within a controlled range. The need for this module is

mainly for efficiency. It would be possible to implement this as a knowledge base but for algorithmic tasks such as this, a dedicated process is more suitable. The data comparator links with the same fact base modules as the inference engines and accesses them in the same way, although its function is performed purely algorithmically.

There will be inevitable variations between the engine and the simulator that may distort the information received by the inference engines. It would, for example, be possible for the rule bases to regard an error of 10% in one sensor reading to be healthy (because this is a known area of deviation) but it is more sensible for this knowledge to be held by the data comparator. It should be capable of normalising the data sources so the EDS itself sees a true deviation figure. This removes the need for complex tuning of the rule bases and makes it much simpler to apply the system to new engines as only one module needs to be changed.

As an aside, the data comparator can also be used to help calibrate the engine simulation. This will be particularly useful when applying the system to new engine models when a number of empirical factors have to be determined.

8.1.5 Knowledge Acquisition

A crucial part of the system is the domain knowledge itself. The way that this is obtained and represented will determine the eventual performance of the system. Although it is possible for an expert system to consistently and quickly apply domain knowledge to a problem, it is ultimately the quality of this knowledge that determines the quality of the system.

Knowledge can be acquired in several ways. The process usually involves a knowledge

engineer (the person responsible for building the knowledge bases) who will interview the domain expert to elicit the expert's knowledge. The KE must be acquainted with the subject area of the expert system so there is an initial learning phase in which the KE obtains sufficient background knowledge to allow him to ask the expert sensible questions. At this stage the expert should also be acquainted with expert system concepts so he will be able to tailor his responses to better suit the KE.

After the initial learning phase the KE should be able to converse about the problem domain in the language that the expert himself understands. The domain specific language will form an important part of the final system as the user interaction should use this language. At this point, a number of sample problems should be considered. Solving sample problems is a way for the expert to learn how to express his knowledge and for the KE to begin to decide how the system should be implemented.

The sample problems should be well understood cases that the expert is well able to solve. In addition, they should cover a range of the problem space so that the KE can gauge the range of knowledge that he will need to represent. The knowledge acquisition process can now begin to take place. Normally a number of interviews will be held in which the KE will try to find out how the expert solves problems. In many cases, the expert will find this process difficult because he will work on a level above this—he will simply know what to do, rather than making conscious decisions about appropriate actions. It is for this reason that the sample problems must be well understood so that the KE is able to probe the expert for more details about his reasoning process. From these interviews the KE can start to formulate the knowledge representation. This is difficult because he must decide on a representation to some extent before he knows what he needs to represent but the background work and the representative problem set

will aid this process. The most important thing is to start to build the prototype during the knowledge acquisition process so that the gaps in the system's knowledge can be identified. If the first attempt at representing the knowledge proves to be wrong, a new scheme can be used but it is important that the knowledge is represented in some form immediately.

It is normally wrong to try to elicit knowledge in a breadth-first manner to gain a general overview of the problem space and then refine this gradually to a lower and lower level. The expert never uses his knowledge like this—he will usually solve a problem depth first. Human memory is highly associative so following the natural course of deductions and actually solving problems is essential to obtain the maximum information from the expert. His memory and diagnostic skills will simply not work out of context or in a random access manner.

Rapid prototyping of expert systems is essential. The test problem cases should be built into a prototype system *during* the interview stage. This helps to show the expert how his knowledge is being used and also highlights the areas which have not yet been sufficiently covered. After the prototyping stage is complete, the system proper can be built. At this stage, the KE and the expert should understand their roles in the process and be able to converse freely about the subject.

8.2 Building An Engine Diagnostic System Using Grape

8.2.1 Using the Rule and Fact Languages

Designing applications for a real-time diagnostic system takes some special care. Unlike normal consultative expert systems, the EDS has no final conclusion but rather continues forever, occasionally indicating that faults are present or that some preventative action should be taken. This section shows how the rule and fact base languages are used when building a new expert system.

The fact bases must be implemented first. The rule base compiler expects to be able to find the fact bases to which the rules refer so it can perform the necessary validation and type checking. The fact bases are formed from a number of object definitions of the form:

```
object cylinder ("Engine Cylinder")
{
  int   capacity ("Cylinder volume (cc)") = 1833;
  float pressure ("Pressure (bar)");
  string condition ("Condition Summary") = "normal";
}
```

Each object describes a set of (related) facts and may specify default values if required. The use of description fields for each entity enables the system to interact with human users in a more coherent manner. In addition to the forms shown, variables can also be declared for trend analysis by preceding the type with the keyword `trend`. An example of trend analysis is shown later. The declaration of objects does not, itself, define any instances of this object. Typically the fact base will also contain a number of instances of certain objects and may also define the values of instances to be different from the default. For example

```
cylinder cyl1;
cylinder cyl2;
cyl1(condition) = "worn";
```

Notice how the facts can be referred to explicitly. In this case two instances have been created and the value of one field has been changed (from the default). Not all instances have to be defined in this way. These static instances are stored such that rules can access them directly making them particularly efficient to reference. Often however, this predeclaration is not possible (or desired) and dynamic instances can be created at run time, using the `add` command.

Each rule base contains references to a set of fact bases which are used by the rules. These references take the form of `include "fbname"`. Note that this is not `#include` as that would be expanded by the C pre-processor. All fact and rule base sources are pre-processed allowing C style comments to be used, macro definitions to be declared and the use of the other pre-processor facilities. The fact bases should be compiled before the rule bases as the rule base compiler accesses the compiled form of the fact bases.

Each rule is formed from a set of variable declarations, a set of conditions and a set of actions. A simple rule would look like

```
rule("Example Rule")
{
  cylinder ?c;
  if
    ?c(condition) != "normal";
  then
    printl "There is a problem with "
    print  ?c.name;
}
```

This rule is simple yet quite powerful. The variable `?c` is declared to be of a certain type

and before the `if` statement, it represents *all* instances of that object. The condition statement compares each of the values of the variable as directed and provided that some statements are satisfied, these successful bindings will proceed to the actions. In this case, this rule will result in a message for *every* cylinder instance for which the condition is not normal.

The rule base language allows a number of other facilities. Dynamic instances can be created and destroyed as required allowing a form of message passing between rule. For example, consider the following rules.

```
rule("Client")
{
  cylinder ?c;
  if
    ?c(temperature) > MAX_CYL_TEMP;
  then
    add message ?m;
    ?m(type) = OVERHEATING;
    ?m(data) = CYLINDERS;
}
rule("Server")
{
  message ?a;
  if
    /* OVERHEATING is defined in a C style header.
       This keeps the code readable but allows
       'type' to be an int */
    ?a(type) == OVERHEATING;
  then
    /* Do some appropriate action */
    ...
    remove ?a;
}
```

In this scenario, the first rule creates a new instance of a particular type when it wants some work to be performed. The second rule, that would be running in a separate process, is servicing all the work requests of this particular type.

8.2.2 Splitting the Rule and Fact Bases

Grape is a parallel system intended to run a number of inference engines simultaneously to solve large complex problems. The way in which the knowledge is split into the individual fact bases and rule bases depends on a number of factors.

- Fact bases are the granularity at which data can be locked. Any data that resides in a fact base that is locked, but is not actually used during the lock, is made unavailable unnecessarily. If this value is private to the inference engine that locked the fact base then this does not matter but if it is shared data, it may cause unnecessary blocking. As with any parallel system, the data dependencies between different inference engines must be studied closely.

- Dividing the rule bases is to some extent a simpler task although grain size and communication cost must be considered. There is inevitably an overhead in creating and maintaining an additional process so rule bases should not be split to such a degree that work loads are less than or of a similar order to the overhead of process creation. Locality is fundamental to all parallel systems. It is essential that rule bases can spend most of their time accessing local fact bases and only have to access remote fact bases occasionally. This again determines the distribution of the fact bases themselves.

- If possible, the rule bases should form logical units rather than simply being a subset of a large rule set. It is better to divide the problem domain and create rule bases for the different areas of the problem. This allows more parallel execution and is also a more natural way for the programmer to work.

- Parallelism can be exploited by the use of rule base splitting allowing independent rules to be executed on different processors. Some applications will have a number of techniques that may be applied to solve a given problem and these are best implemented in separate rule bases so they may be tried in parallel.

- There is no provision as such for local scratch fact bases that are not shared. These can be created simply however by replicating the existing fact bases so a unique name exists. This is a very useful technique especially when a number of workers are working on the same problem.

8.2.3 Uncertainty and Missing Data

In any system that is dealing with real data, it is inevitable that data will be unreliable. Sensors may malfunction, wiring become unsound or the data acquisition boards themselves could develop faults. Other transient problems such as electro-magnetic interference can also cause data values to become distorted. More fundamentally, the sensors may have inherently high error rates or give readings within a wide tolerance of the actual value. These problems must be addressed if a reliable diagnostic system is to be produced.

Inherent data noise is dealt with by the uncertainty mechanism of the inference engine. The data values are given a suitable confidence factor that ensures their impact on the diagnosis reflects the confidence in the sensor reading.

Another approach is to have a data validation stage in the data acquisition system. This method can provide the necessary screening of intermittent bad data readings but would have the ability to recognise when the sensor has actually failed or if the data

has really changed significantly. A simple implementation of this would be given the acceptable range of data from each sensor. Intermittent data reliability problems can be dealt with by either ignoring single data values that are not similar to previous values or by using an average of the last few readings. Sensor faults can be detected if the data falls out of the acceptable range but more subtle reliability problems could also be notified by storing the variance of data readings over time. The exact structure of such a system would have to be determined using runs of real engine data over an extended period of operation and as such has not been undertaken in this project. More analytical techniques such as state estimation could also be used to predict the expected values of sensors and warn the system and the operators of any potential problems.

8.2.4 Reasoning with Time

The rule and fact base languages allow very powerful systems to be created. In addition to simple values, each instance has an uncertainty value and a recency number that are maintained by the system. Each variable (and each assignment) can be given an optional uncertainty value. Each value is also time-stamped with its time of creation. These time-stamps can be accessed directly by the rule bases allowing them to reason about time and to monitor the performance of other inference engines. The time-stamps of data values are accessed using the following construction.

```
cyl1(temperature).time
```

In general, the `time` field can be replaced by `odds`, `desc` or `value` (which is the default) depending on what is required.

8.2.5 Trend Analysis

Facts may also be declared to hold a specified (10 by default) number of old values which is particularly useful for trend analysis. Consider the following code.

```
object cylinder ("Engine Cylinder")
{
  ...
  trend 5 int temperature ("Temperature (c)");
}
...
rule ("Falling Temperature")
{
  cylinder ?c;
  if
    ?c(temperature[0]) < ?c(temperature[1]);
    ?c(temperature[1]) < ?c(temperature[2]);
  then
    /* Take appropriate action */
    ...
}
```

The cylinder temperature variable is declared to store four previous values (in addition to the current value). Each value may be referenced individually and each individual value has a distinct time-stamp and odds value. In the above example, the action is only performed if the cylinder temperature is falling and was previously falling. The values are stored in a ring buffer and new values push the previous values further down the list. In this way, the zero index is always the current value.

8.3 Making a Prototype System

In the time scale of this project, it was not possible to build a prototype EDS so a small test system was constructed to test Grape itself. The next phase of the development is to build the first prototype EDS that can form the nucleus of the full system.

8.3.1 Knowledge Acquisition

For the test rule bases, the knowledge was acquired manually from simulated engine performance data. A small number of faults were modelled and the engine performance was plotted for each case. By inspection, some fault characteristics were extracted to represent the different fault categories. This is a simple method for acquiring knowledge and is adequate for a system designed mainly to test the Grape system itself but it is a dangerous method if applied to a real diagnostic system. The process of analysing the graphs should be performed by an expert who understands the complex physical system under test. Diagnosing faults from engine sensor data is not something that even experienced engineers or mechanics do (they use more direct methods of fault diagnosis), but they will be much more capable of extracting the relevant information from performance data.

As the simulation has been validated against the real engine, this data can be used to help build the knowledge base and to test the prototype system. However, the simulation must not be used alone—it is essential that real engine data is also used as much as possible to validate the system (and the simulation).

The knowledge bases for the prototype will be small and probably not need to be split at all. However, as the system grows into the full system, a multiple rule base system will be required. It is advisable therefore to split the rule bases immediately into the projected functional units of the EDS. For example, it may be sensible to have a top level supervisor process to monitor the engine and spawn specialist workers for diagnosing particular faults. Rule bases could exist for cylinder faults, timing faults, fuel faults etc. and be called as necessary.

8.3.2 Fault Selection

Selecting faults for the prototype system is important if it is to achieve its aim of showing the validity of the approach and forming the nucleus of the full system. The faults must be representative of the problem domain so that the potential difficulties can be identified as soon as possible.

Only a small number of faults are needed initially but they should have a number of properties. Firstly, the simulator should be capable of modelling the faults. The simulator is currently limited to some extent in the range of faults that can be modelled so it was important to select faults for which simulated data could be generated. Secondly, the faults should be divisible into sub-groups so that the envisaged use of a hierarchical diagnostic model can be tested. Finally, it should be possible to introduce these faults to the real engine simply and without causing any damage. This will be important for verifying the system at a later date.

A possible first set of faults may include the following five conditions. An advanced injection angle; a retarded injection angle; a compressor fault; a turbine fault and a healthy engine. The healthy engine should be included as a control case because clearly the system must be able to detect when the engine is healthy.

8.3.3 Data Sources

The prototype system must have access to both simulator and engine data. The engine, in the Department of Mechanical Engineering, is instrumented and connected to an IBM-PC based data acquisition system, using commercial analogue to digital (A to D)

converter boards. The data is read from these boards into a spreadsheet package. In an system installed on board ship, this hardware would be replaced by dedicated A to D hardware mapped directly into the address space of the transputer system.

As the engine is in use by others and the engine sensors are already connected to the PC-based data acquisition system, the simplest solution is to use the existing acquisition system to log data (with time-stamps) to disk and then simulate the engine using this stored information. The data acquisition process in the prototype EDS simply needs to read in this file and generate data according to the timing information stored in the file. To the rest of the system, it appears as if a real engine is connected but allows for experimentation without the cost and inconvenience of actually running the engine each time. This approach is quite suitable for prototyping as the system as it is possible to re-apply the fault conditions many times without needing to re-run the engine. Especially for faulted conditions where the engine is working outside its normal behaviour, it is an advantage not to have to repeat these tests too frequently. It should never be overlooked however that the real EDS must be validated against real engine data generated on-line.

The simulator data can be acquired in a similar way to the engine data to allow more processors to be free for the EDS itself. In practice, two racks of processors are required, one to run the engine simulation and one (possibly smaller) to run the EDS. To reduce this demand during prototyping, data can be saved with timing information and retrieved by a process that appears to the rest of the system to be the simulator. As the current simulator is unable to run in real-time, this approach allows the data to be injected into the system in real-time, and therefore in step with the real engine data.

Data Cleanliness for Simulated Reference

Real engine data will be noisy and to some extent unreliable. If a simulator is to be used for the development of an EDS, then this behaviour must also be simulated so the robustness and sensitivity of the system can be assessed. Simulating realistic sensors can be achieved by adding a filter module to the input of the data comparator. The module must be able to add random noise to the input data (preferably in the same bandwidth as the real sensors) and possibly also simulate the high tolerance or unreliability of sensors. To some extent this modelling can become arbitrarily complex but it is always preferable to use genuine engine data if it is available. A good noise 'filter' is however very useful as it immediately provides a vast amount of test data without the need to run real engine tests.

8.3.4 Sensor Selection

The diesel engine used in this project, located in the School of Mechanical Engineering at the University of Bath, has been instrumented for both automatic diagnostic work and for other diesel engine research. The engine has a great number of sensors including ones to measure the temperature and pressure of the engine gases at various positions in the engine as well as speed and torque meters, an angle sensor and a top dead centre (TDC) detector. Each of these sensors could be used to derive data on which to base the diagnosis but this level of instrumentation is unrealistic for engines in service. Due to the cost and maintenance of more complex sensors (such as in-cylinder pressure sensors), much of this data would not be available to a real diagnostic system. Some of these values can be derived using simpler techniques or non-intrusive sensors (such as

magnetic sensors to monitor pistons) but these methods are still the subject of research. For these reasons, only a small number of simple sensors were used as the input to the EDS. For realistic systems, a subset of these sensors should be found that could be fitted to engines for a reasonable cost and that would not require constant maintenance.

8.4 Summary

This chapter has described the design of a diesel engine diagnostic system using the Grape system. Each component of the design has then been discussed individually. The use of the rule and fact base languages is also described. The method of building an EDS on top of Grape is discussed and the development of a prototype system is proposed. Only a small test system has been built to date although many of the facilities required by a full system have been implemented.

Chapter 9
Further Work

9.1 Further System Development

The Grape system is fully functional in its present state. There are however a number of features that could be added to the system if they became necessary. Two features would be likely to be of benefit in a wide variety of situations: active knowledge bases and a knowledge-based scheduler.

The knowledge bases should be active in the sense of a more conventional blackboard system. Active blackboards allow actions to be associated with particular facts in the knowledge base so that the act of changing the fact base causes some function to be performed. This is relatively straight forward to implement and would lead to significant improvements in system efficiency under certain conditions. Firstly some simple inference engine functions can be discarded completely with the addition of even a limited data activity. More importantly, tasks could block efficiently on certain data items which currently require spin-locks or test-sleep-test cycles. With this facility a powerful filter based reasoning system could be implemented giving more control of the inferencing process at a higher level.

Another interesting addition would be to use the speed and power of the inference engines to implement a knowledge-based scheduler. It has been shown that optimal

schedules are impossible to achieve but often there is other information known about the task set which could be used to direct the scheduling process. If knowledge exists of likely process use or of particular system behaviour, a knowledge based scheduler may be able to perform better than a purely algorithmic one. It would be possible to provide an interface that allows a developer to design a rule base to assist in the scheduling process. The use of a knowledge-base scheduler would require knowledge about the task set and also significant development time so the algorithmic scheduler would always be available for normal use or to take over under certain conditions.

9.1.1 Parallel Computer Support

The majority of the developed system is extremely portable. The individual inference engines, the local scheduler and most other modules are directly portable to any system with standard C and C++ compilers and very basic operating system support. The one area which remains system specific is the support for parallelism.

As parallel computers become more common, it is clear that a standard approach must be established to program these machines. Software companies which until recently have been able to write dedicated software for individual clients now find that a far wider group of users has access to parallel computers. To avoid the waste of resources in porting individual software packages to each new architecture, some standardisation must exist. If a standard library exists then only one person need port the library to each new architecture—the existing software can then just be recompiled (or even just re-linked).

There are a number of separate 'open standards' being proposed at the current time but

little software exists to support these. One notable exception is the p4 system developed at the Argonne National Laboratories [99]. p4 is a programming model designed primarily for shared memory architectures but that has more recently been extended to distributed memory systems. It provides a set of standard functions including synchronisation, communication and process spawning that can be used to develop a portable parallel program. The source code is then linked with the appropriate p4 library depending on the target platform required. Currently, systems such as the Sequent Symmetry and Encore Multimax as well as networks of Sun workstations and the new KSR-1 machine from Kendall Square Research are supported. The library is easy to port to new systems and can provide a means of developing more portable parallel code in the future.

It would be a worthwhile exercise to port the p4 system to the Bath transputer system. It would allow the development of more portable software as well as allowing other software to be more easily ported to it.

An interesting development would be to port Grape to a virtual shared memory machine. Such machines allow the use of the existing shared memory paradigm but give far greater scalability than bus-based architectures. Some VSM implementations such as the Data Diffusion Machine [100] have no fixed home locations for data but allow it to migrate around the system according to its use. Such a machine would distribute the shared fact bases over the machine and provide an efficient mechanism for fact base sharing. In the current version of Grape, the application programmer must make some decisions, based on the locality of data, about which data is stored in which fact bases and how these should be shared. A cache only memory architecture, although not removing the need to consider locality, would remove this burden to some extent

by distributing individual fact bases over the machine depending on their current use.

9.1.2 The User Interface

In a commercial system, the user interface is very important for a number of reasons. Any product should be as clear and simple to use as possible and computer software is no exception. For development purposes, Grape's user interface has been very basic, using a standard text screen for both input and output. To prepare the system for a commercial, or non-developmental platform, a more user friendly interface would need to be implemented. The introduction of new technology can be very difficult especially in areas that have to date resisted the use of new equipment and the choice of human-computer interface (HCI) can determine a system's acceptance. A skilled engine mechanic will be more likely to accept a machine's opinion if it can be expressed in the language and diagrams that he/she would use.

A great deal of work has been done in recent years on implementing effective HCI's and much of this work is in the public domain. There are a number of graphics libraries available specifically for implementing HCI's and these in general support a number of target platforms. For portability in general and specifically under Helios and Unix any chosen HCI library must support X Windows.

9.2 Knowledge Acquisition

So far, only a small test knowledge base has been built for engine diagnostics. The next stage of work should be to use Grape to build a prototype system with the help of at least one domain expert. Once verified, the prototype knowledge base can be expanded

to a full range of fault conditions as well as incorporating condition monitoring, alarms and even running-point optimisations. A significant time would need to be spent in developing and testing the knowledge base but it would provide an extremely powerful tool that would be able to contribute to very large financial savings and increased safety.

Connection to a live diesel engine to generate real-life test data is essential. The diesel engine simulation can be used to aid knowledge acquisition and to help test the system (as it has been verified against a working engine by both Haysom [4] and Shamail [2]) but real data would have to be used if there was to be general confidence in the system.

A framework now exists in which large, concurrent knowledge bases can run efficiently and securely on a parallel computer. The system allows very complex reasoning processes to be developed while still maintaining the real-time necessity of engine diagnostics.

9.3 System Integration

The prototype EDS has not yet been interfaced to the outside world in real-time. Instead, real world data (and simulation data) is time-stamped and stored on disk. Data acquisition modules are used to collect this data and feed it in to system as if the data were 'live'. To integrate the system to the real world requires some simple data acquisition hardware and some modifications to the current data acquisition software. This has not been implemented because of time constraints and the lack of access to live data sources.

9.4 Use of the Real-Time Diesel Engine Model

Clearly, a condition monitoring system must have a reference with which to compare the real engine data. As mentioned earlier, most solutions to this problem rely on using approximate interpolation (or even extrapolation) of stored engine maps. An accurate and sufficiently fast simulation is able to provide this comparison data in a very efficient manner.

In prototyping a diagnostic system, the real-time model can be used for knowledge acquisition and for the generation of test cases. Beyond this, the model could be utilised in a more direct way. The simulation capability will increase naturally with time as processing power increases and this will in turn increase its potential uses.

As the fault diagnosis proceeds, the inference engines generate a set of likely fault conditions ranked according to the system's confidence in them. It can be difficult in practice to differentiate between faults that give similar symptoms and this becomes even more difficult when multiple fault patterns are considered. A powerful use of the engine simulation would be to test a set of possible fault hypotheses. Each fault candidate could be modelled on the simulation and the simulator outputs compared with those being obtained from the engine. If the simulation was sufficiently fast, such a processes could greatly increase the accuracy of diagnosis. This subject will require a great deal of research but the Grape framework will support such a system.

Chapter 10
Conclusions

10.1 General

This work has resulted in the development of a general purpose real-time parallel knowledge-based system (Grape). The portability of the system and its freedom from any particular problem domain makes it a useful tool for a wide variety of time critical systems where it would be desirable to apply a knowledge-based approach.

A system of co-operating inference engines is used which provides a responsive and dynamic system. Each inference engine accesses a single rule base and a number of (possibly shared) fact bases. The sharing of facts is done in a secure way with each fact base containing a lock that may be used to enforce mutual exclusion. The fact bases may be distributed amongst the processors and may even move during the lifetime of the system as the processes accessing it are scheduled on other processors. Each data item is time-tagged allowing the inference engines to reason about the time sequence of events. Each individual inference engine operates very quickly and is able to evaluate a typical rule in between 80 and 1000μs (depending on rule complexity) on a single T800 transputer.

Two compilers have been developed, one for the rule language and one for the fact base definition language. The fact bases are specified in an object oriented manner in which

related groups of facts may be collected into single objects. This approach provides a more natural representation of knowledge and hence eases the task of knowledge base building. The rules are written in another dedicated language that provides a flexible means of manipulating and reasoning with the fact bases. Rules may contain explicit facts (such a `cylinder1`) or use variables to represent classes of facts (such as `cylinder ?c`). The ability to reason about sets of facts in a single rule allows powerful concise rule bases to be constructed. As well as altering the fact bases directly, rule actions may also request user action, spawn new tasks, control the priorities of other tasks in the system etc. Facts (data values) are stored with time-stamps so that the rules may reason about the time and facts may additionally be declared to store a number of trend values for trend analysis.

10.2 Scheduling

A number of inference engines can be executed concurrently either on a single processor or on a number of processors. A scheduler on each processor is used to control the resource usage of each process, to schedule the processes according to their priority and their temporal constraints and to perform the multitasking needed to run multiple tasks on a single processor.

The scheduler is an important aspect of any real-time system having ultimate control over the temporal characteristics of the system. For this project, a co-operative local scheduler has been designed that enables processes to be controlled in a portable manner and more fundamentally, allows time critical scheduling to be performed under Unix and Helios. The co-operative scheduler has been enhanced to allow other processes

be scheduled during disk accesses etc., resulting in increased processor utilisation.

For multiprocessor scheduling, a two phase scheme has been used. A local scheduler on each processor attempts to guarantee new processes as they arrive. This local guarantee is performed by a new algorithm called the Slack Time List (STL) algorithm. The STL algorithm is sub-optimal but has been shown to perform better than a more expensive optimal algorithm under most conditions. The speed of the STL algorithm makes it particularly suitable for use in multiprocessor systems where decisions of schedulability have to be made quickly. Tasks that are rejected by the STL are then migrated to other processors according to the global scheduling scheme. A number of global policies were investigated and scheme using a form of focussed addressing was selected as it performed better on average than the other schemes tested. The best algorithm depends however on the application to some extent so the facility of selecting algorithms is provided. It may be that on systems with a different communication to computation ratio (notably, more expensive communication) that the bidding scheme will prove to be more efficient.

10.3 Fault Diagnosis

The design for a diesel engine fault diagnosis system is presented. The implementation of each element of the system is discussed. The use of the rule base and fact base languages is shown, with attention to details of interest to building a real-time diagnostic system. Only a small test system has been built to date to test the functionality of the Grape system and this must be extended into a larger prototype system and tested. The prototype can then be expanded into a full system and used in practice.

References

[1] Lilley, L. C. R. *Diesel Engine Reference Book.* Butterworths, London, 1984.

[2] Shamail, S. *Distributed Memory Diesel Engine Simulation Using Transputers.* PhD thesis, University of Bath, 1990.

[3] Jones, A. D. *The application of parallel processing to diesel engine modelling.* PhD thesis, University of Bath, 1987.

[4] Haysom, F. J. *Enhanced Performance Simulation of Diesel Engines.* PhD thesis, University of Bath, 1989.

[5] PAYDIRT. *Processing Architecture Yeilding Dedections in Real-Time: PAY-DIRT.* Technical Annex to ESPRIT II Project 5483, 1990.

[6] Gondran, M. *An Introduction to Expert Systems.* McGraw-Hill, 1986.

[7] Ernst, G. W. and Newell, A. *GPS: A case study in Generality and Problem Solving.* Academic Press, New York, 1969.

[8] Newell, A. *Unified Theories Of Cognition.* Harvard University Press, 1990.

[9] Lyndsay, R. K., Buchanan, B. G., Feigenbaum, E. A. and Lederberg, J. *Applications for Artificial Intelligence for Organic Chemistry.* McGraw-Hill, 1980.

[10] Shortliffe, E. *Computer Based Medical Consulatations: MYCIN.* American Elsevier, New York, 1976.

[11] Buchant, J. and McDermott, J. R1 revisited after four years in the trenches. *AI Magazine*, 5:21–32, 1984.

[12] Duda, R., Gaschnig, J. and Hart, P. Model Design in the PROSPECTOR Consultant System for Mineral Exploration. In Michie, D., editor, *Expert Systems in the Micro-Electronic Age.* Edinburgh University Press, 1979.

[13] Findler, N., editor. *Associative Networks: Representation and Use of Knowledge by Computer.* Academic Press, New York, 1979.

[14] Minsky, M. A Framework for Representing Knowledge. In Winston, P., editor, *The Phychology of Computer Vision.* McGraw-Hill, New York, 1975.

[15] Lloyd, J. W. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[16] Gupta, A. *Parallelism in Production Systems.* Pitman, London, 1987.

[17] Forgy, C. L. *On the Efficient Implementation of Production Systems.* PhD thesis, Carnegie-Mellon University, Pittsburgh, 1979.

[18] Forsyth, R., editor. *Expert Systems: Principles and Case Studies.* Chapman and Hall Computing, London, 2nd edition, 1989.

[19] Sell, P. S. *Expert Systems: A Practical Introduction.* Macmillan, 1985.

[20] Qi, C. Y. A Two-Stage Inference Strategy for Real-Time Expert Systems. *Expert Systems with Applications*, 1:179–182, 1990.

[21] Laffey, T. J., Cox, P. A., Schmidt, J. L., Kao, S. M. and Read, J. Y. Real-Time Knowledge-Based Systems. *AI Magazine*, Spring, 1988.

[22] Verbruggen, H. B., Honderd, G. and Bruijn, P. M. Expert Systems in Real-Time Control and Monitoring. In *Power High Tech. Conference*, Valencia, 1989.

[23] Masui, S., McDermott, J. and Sobel, A. Decision Making in Time Critical Situations. In *Procs. 1983 International Joint Conference on Artificial Intelligence*, pages 233–255, 1983.

[24] Gustafson, J. L. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, May 1988.

[25] Ishida, T. Parallel Rule Firing in Production Systems. *IEEE Transaction on Knowledge and Data Engineering*, 3(1), 1991.

[26] Li, T., Fang, L. and Wilson, W. H. Running rule-based expert systems on parallel processors. *Knowledge Based Systems*, 2(1), 1989.

[27] Fennel, R. D. and Lesser, V. R. Parallelism in Artificial Intelligence Problem Solving: A case study of Hearsay II. *IEEE Transactions on Computers*, C-26(2), 1977.

[28] Shaw, D. E. NON-VON: A Parallel Machine Architecture for Knowledge-Based Information Processing. In *Procs. 1981 International Joint Conference on Artificial Intelligence*, pages 961–963, 1981.

[29] Rieger, C., Trigg, R. and Bane, B. ZMOB: A New Computing Engine for AI. In *Procs. 1981 International Joint Conference on Artificial Intelligence*, pages 955–960, 1981.

[30] Stolfe, S. J. and Shaw, D. E. DADO: A Tree Structured Machine Architecture for Production Systems. In *Proc. National Conference on Artificial Intelligence*, 1982.

[31] Togai, M. and Watanabe, H. Expert System on a Chip: An Engine for Real-Time Approximate Reasoning. *IEEE Expert*, 1(3), 1986.

[32] Sharma, D. D. and Sridharan, N. S. Knowledge-Based Real-Time Control: A Parallel Processing Perspective. In *Procs. of American Association of Artificial Intelligence Conference*, pages 10–20, 1988.

[33] Pazirandeh, M. and Becker, J. Object Oriented Performance Models with Knowledge-Based Diagnostics. In *Proc. Winter Simulation Conference*, 1987.

[34] Gonzalez, R. C., Fry, D. N. and Kryter, R. C. Results in the application of pattern recognition methods to nuclear reactor core component surveillance. *IEEE Transactions on Nuclear Science*, 21(1):750–757, 1974.

[35] Cardozo, E. and Talukdar, S. N. A Distributed Expert System for Fault Diagnosis. *IEEE Transaction on Power Systems*, 3(2), 1988.

[36] Keravnou, E. T. and Johnson, L. Towards a generalized model of diagnostic behaviour. *Knowledge Based Systems*, 2(3), 1989.

[37] Thompson, W. B., Johnson, P. E. and Moen, J. B. Recognition Based Diagnostic Reasoning. In *Procs. 1983 International Joint Conference on Artificial Intelligence*, 1983.

[38] O'Leary, J. P. New Opportunities in diesel condition monitoring. *Institution of Diesel and Gas Turbine Engineers*, 1988. Discussion document.

[39] Katsoulakos, P. S., Newland, J., Stansfield, J. T. and Ruxton, T. Monitoring, Databases and Expert Systems in the Development of Engine Fault Diagnostics. *British Journal of Non-Destructive Testing*, July 1988.

[40] Katsoulakos, P. S., Hornsby, C. P. W. and Zanconato, R. DEEDS: The Diesel Engine Expert Diagnosis System. *Maritime Communications and Control*, October 1988.

[41] Shamsolmaali, A. and Banisoleiman, K. Real time diagnostic techniques and applications. In *Enhanced Safety and Profitability in the Maritime Industry*, March 1991.

[42] Hoare, C. A. R. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[43] Dijkstra, E. W. Cooperating sequential processes. In Genuys, F., editor, *Programming Languages*. Academic Press, New York, 1968.

[44] Levi, S.-T. and Agrawala, A. K. *Real-Time System Design*. McGraw-Hill, 1990.

[45] Allworth, S. T. *Introduction to Real-Time Software Design.* Macmillan Press, 1981.

[46] Stankovic, J. A. Real-time computing systems: the next generation. In Stankovic, J. A. and Ramamritham, K., editors, *Tutorial: hard real-time systems*, pages 14–38. IEEE, 1988.

[47] Liu, C. L. and Layland, J. W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the Association of Computing Machinery*, 20(1):46–61, 1973.

[48] Dertouzos, M. L. and Mok, A. K.-L. Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks. *IEEE Transactions on Software Engineering*, 15(12), 1989.

[49] Sha, L., Lehoczky, J. P. and Rajkumar, R. Task scheduling in distributed real-time systems. In *Proc. IEEE Industrial Electronics Conference*, 1987.

[50] Mok, A. K.-L. *Fundamental design problems of distributed systems for hard-real time environments.* PhD thesis, Michigan Institute of Technology, 1983.

[51] Garey, M. R. and Johnson, D. S. *Computers and Intractability.* W.H. Freeman and Company, 1979.

[52] Horn, W. A. Some simple scheduling algorithms. *Naval Research Logistics*, 21, 1974.

[53] Houstis, C. E. Module Allocation of Real-Time Applications to Distributed Systems. *IEEE Transactions on Software Engineering*, SE-16(7), 1990.

[54] Sarje, A. K. and Sagar, G. Heuristic model for task allocation in distributed computer systems. *IEE Proceedings-E*, 138(5), 1991.

[55] Chu, W. W. and Lan, L. M.-T. Task Allocation and Precedence Relations for Distributed Real-Time Systems. *IEEE Transactions on Computers*, C-36(6), 1987.

[56] Smith, R. G. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, C-29(12), 1980.

[57] Alijani, G. S. and Wedde, H. F. Enhanced Reliability in scheduling critical tasks for hard real-time distributed systems. *Lecture Notes in Computer Science*, 497, 1991.

[58] Stankovic, J. A., Ramamritham, K. and Cheng, S. Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems. *IEEE Transactions on Computers*, C-34(12), 1985.

[59] Zhao, W., Ramamritham, K. and Stankovic, J. A. Scheduling Tasks with Resource Requirements in Hard Real-Time Systems. *IEEE Transactions on Software Engineering*, SE-13(5), 1987.

[60] Ramamritham, K., Stankovic, J. A. and Zhao, W. Distributed Scheduling of Tasks with Deadlines and Resource Requirements. *IEEE Transactions on Computers*, C-38(8), 1989.

[61] Zhao, W., Ramamritham, K. and Stankovic, J. A. Preemptive Scheduling Under Time and Resource Constraints. *IEEE Transactions on Computers*, C-36(8), 1987.

[62] Stankovic, J. A. Decentralized Decision Making for Task Reallocation in a Hard Real-Rime System. *IEEE Transactions on Computers*, C-38(3), 1989.

[63] Stankovic, J. A. and Ramamritham, K. The Spring Kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3):62–72, 1991.

[64] Stankovic, J. A. Stability and Distributed Scheduling Algorithms. *IEEE Transactions on Software Engineering*, SE-11(10), 1985.

[65] Thompson, K. UNIX implementation. *The Bell System Technical Journal*, 57(6), 1978.

[66] Lycklama, H. and Bayer, D. L. The MERT Operating System. *The Bell System Technical Journal*, 57(6), 1978.

[67] Verhulst, E. Preemptive process scheduling and meeting hard real-time constraints with TRANS-RTXc on the Transputer. In *Procs. 2nd International Conf. on Applications of Transputers*, July 1990.

[68] P1003.1. *IEEE P1003.1 Portable Operating System Interface for Computer Environments (POSIX)*. Institute of Electrical and Electronic Engineers, Piscatsway, NJ, 1988.

[69] Noronha, S. J. and Sarma, V. V. S. Knowledge-Based Approaches for Scheduling Problems: A Survey. *IEEE Transaction on Knowledge and Data Engineering*, 3(2), 1991.

[70] Thesen, A., Yih, Y. and Lei, L. Knowledge Acquisition Methods for Expert Scheduling Systems. In *Proc. Winter Simulation Conference*, 1987.

[71] Johnston, M. D. and Adorf, H.-M. Scheduling with neural networks—the case of the Hubble Space Telescope. *Computers Operations Research*, 19(3/4):209–240, 1992.

[72] Holland, J. H. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, 1975.

[73] Davis, L. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

[74] Goldberg, D. E. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley Publishing Company, Inc., 1989.

[75] Syswerda, G. Schedule Optimization Using Genetic Algorithms. In Davis, L., editor, *Handbook of Genetic Algorithms,* pages 332–349. Van Nostrand Reinhold, 1991.

[76] Tanner, D. G. *Real time simulation of power systems.* PhD thesis, University of Bath, 1982.

[77] Williams, S. K. *Power system optimisation and stability studies using real-time simulation.* PhD thesis, University of Bath, 1986.

[78] Dale, L. A. *Real time modelling of multi-machine power systems.* PhD thesis, University of Bath, 1986.

[79] Berry, T. *Real time simulation of complex power systems using parallel processors.* PhD thesis, University of Bath, 1989.

[80] Dunn, R. W., Daniels, A. R., Gott, V. S. and Selwyn, C. G. A new architecture of high performance parallel computer for use in condition monitoring of large diesel engines. In *Procs. 2nd International conference on software engineering for real-time systems,* 1989.

[81] INMOS Limited. *IMS T800 Transputer data sheet.* Aztec West, Almondsbury, Bristol. UK, 1987.

[82] May, D. Occam. *Sigplan Notices,* 18(4):69–79, 1983.

[83] Daniels, A. R., Dunn, R. W., Gott, V. S. and Selwyn, C. G. Real time simulation of diesel engines using the T800 Transputer. In *SERC/DTI Transputer Initiative Workshop on Transputer Development Environments,* 1987.

[84] Philips Components Limited. *16/32 Bit Highly Integrated Microprocessor SCC68070 User Manual,* 1988.

[85] INMOS Limited. *IMSC012 Data Sheet.* Aztec West, Almondsbury, Bristol. UK, 1987.

[86] Stallard, P. W. A. *The development of an input/output controller for use in a multi-processor computer system.* BSc thesis, University of Bath, 1989.

[87] Hafeez, M. *An expandable input/output and graphics system for distributed memory parallel computers.* PhD thesis, University of Bath, 1990.

[88] Philips Components Limited. *Video and System Controller SCC66470 User Manual*, 1988.

[89] Perihelion Software Ltd. *The HELIOS Operating System.* Prentice-Hall, 1989.

[90] Kernighan, B. W. and Ritchie, D. M. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[91] Ritchie, D. M. A Retrospective. *Bell System Technical Journal*, 57(6):1947–1969, 1978.

[92] Ritchie, D. M. The Evolution of the UNIX Time-Sharing System. *AT&T Bell Laboratories Technical Journal*, 63(8):1577–1593, 1984.

[93] AT&T. *The System V Interface Definition (SVID).* American Telephone and Telegraph, Murray Hill, NJ, 2nd edition, 1987.

[94] Leffler, S. J., McKusick, M. K., Karels, M. J. and Quarterman, J. S. *The design and implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1988.

[95] ANSI X3.159-1989. *American national standard for information systems: programming language—C.* American National Standards Institute, New York, 1989.

[96] Charlton, S. J. SPICE User Manual. Technical report, University of Bath, 1986.

[97] Naur, P., Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauguois, B., Wegstein, J. H., van Wijngaarden, A. and Woodger, M. Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 3(5):299–314, 1960.

[98] Stroustrup, B. *The C++ Programming Language.* Addison Wesley, 2nd edition, 1991.

[99] Lusk, E., Overbeek, R., Boyle, J., Butler, R., Disz, T., Glickfeld, B., Patterson, J. and Stevens, R. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc., 1987.

[100] Warren, D. H. D. and Haridi, S. The Data Diffusion Machine—A Scalable Shared Virtual Memory Multiprocessor. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 943–952, Tokyo, Japan, December 1988.

Appendix A

Rule and Fact Base Grammars

This appendix describes the syntax grammar of the knowledge representation languages designed for the rule and fact bases. The grammar is presented in a BNF (Backus-Naur Form) type format [97].

A.1 The Terminal Symbols

A context-free grammar is composed of *symbols*. Some symbols are constructed in turn from other symbols according to rules in the grammar—these symbols are known as *non-terminal symbols*. Symbols which do not further sub-divide are called *terminal symbols*. In the following grammars, terminals are shown in bold and all other symbols are in italics.

Most of the terminal symbols correspond directly to the actual typed input. '**rule**' and '**;**' for example are simple terminals that represent the keyword 'rule' and the statement terminator ';'. The remaining terminal characters are defined as the following regular expressions.

ASSIGN	`{=,+=,-=,*=,/=,&=,|=,^=}`
CMP	`{==,!=,>,>=,<,<=}`
DIGIT	`[0-9]`
FPNUM	`{DIGIT}+"."{DIGIT}*`
ID	`[a-zA-Z_][a-zA-Z0-9_]*`
INTNUM	`{DIGIT}+`
TEXT	`\"[^\"]*\"`
VAR	`"?"{ID}`

A lexical analyser reads in the knowledge source code and converts this in to a stream of tokens. These tokens are then parsed by the grammars to ensure that they are

syntactically correct. At that stage, the compilers use their internal code trees to check for semantic errors such as assigning variables of incompatible types.

The keywords used by the rule and fact base languages are **add, desc, float, halt, if, include, input, int, lock, name, object, odds, print, printl, priority, remove, rule, spawn, string, then, time, trend, unlock** and **value**. These words are reserved and cannot be used as identifiers as they are tokenised by the lexical analyser before the compiler is able to assess the context in which they are used.

A.2 The Fact Base Representation

fact base ::= */* Nothing */*
::= *blocks*

blocks ::= *block*
::= *blocks block*

fact ::= **object** *objname* (*objdesc*) *objbody*
::= *instance*
::= *assignment*

objname ::= **ID**

objdesc ::= **TEXT**

objbody ::= { *definitions* }

definitions ::= *definition*
::= **trend** *definition*
::= **trend INTNUM** *definition*
::= *definitions definition*

definition ::= **float** *factname* (*factdesc*) *default* ;
::= **int** *factname* (*factdesc*) *default* ;
::= **string** *factname* (*factdesc*) *default* ;

factname ::= **ID**

factdesc ::= **TEXT**

default ::= */* empty */*
::= = *arg*

instance ::= *objname instname* ;

instname ::= **ID**

assignment ::= *instname* (*factname*) = *arg* ;

arg ::= **INTNUM**
::= **INTNUM** , *odds*
::= **FPNUM**
::= **FPNUM** , *odds*
::= **TEXT**
::= **TEXT** , *odds*

odds ::= **FPNUM**
::= **INTNUM**

Values declared as trend variables will have each element initialised to the same (optionally specified) value. The language does not currently allow the histories to be preset but this feature could be added if required.

A.3 The Rule Base Representation

rulebase ::= */* empty */*
::= *blocks*

blocks ::= *block*
::= *blocks block*

block ::= *ruledef*
::= **include** *fbname*

fbname ::= **TEXT**

ruledef ::= **rule** (*rulename* , *priority*) *rulebody*
::= **rule** (*rulename*) *rulebody*

rulename ::= **TEXT**

priority ::= **INTNUM**

rulebody ::= { **if** *condtions* **then** *actions* }
::= { *vardefs* **if** *condtions* **then** *actions* }

vardefs ::= *vardef*

	::=	*vardefs vardef*
vardef	::=	*objname* **VAR ;**
conditions	::=	*condition*
	::=	*conditions condition*
condition	::=	*expr* **CMP** *expr* **;**
	::=	**lock** *fbname* **;**
	::=	**unlock** *fbname* **;**
actions	::=	*action*
	::=	*actions action*
action	::=	*instance* **ASSIGN** *expression* **;**
	::=	**add** *objname VAR* **;**
	::=	**halt ;**
	::=	**input** *instance* **;**
	::=	**lock** *fbname* **;**
	::=	**print** *instance* **;**
	::=	**print VAR . name ;**
	::=	**print TEXT ;**
	::=	**printl** *instance* **;**
	::=	**printl VAR . name ;**
	::=	**printl TEXT ;**
	::=	**priority** *taskid priority* **;**
	::=	**remove** *VAR* **;**
	::=	**spawn** *taskid args* **;**
	::=	**unlock** *fbname* **;**
objname	::=	**ID**
taskid	::=	**TEXT**
args	::=	**TEXT**
	::=	*args* **TEXT**
instance	::=	*inst*
	::=	*inst* **. desc**
	::=	*inst* **. name**
	::=	*inst* **. odds**
	::=	*inst* **. time**
	::=	*inst* **. value**

inst	::=	*instname* (*factname*)
	::=	*instname* (*factname* [**INTNUM**])
instname	::=	**ID**
	::=	**VAR**
expression	::=	*expr* , *odds*
	::=	*expr*
expr	::=	*instance*
	::=	**INTNUM**
	::=	**FPNUM**
	::=	**TEXT**
	::=	*expr* + *expr*
	::=	*expr* - *expr*
	::=	*expr* * *expr*
	::=	*expr* / *expr*
	::=	*expr* ^ *expr*
	::=	- *expr*
	::=	(*expr*)
odds	::=	**FPNUM**
	::=	**INTNUM**